



AWS Well-Architected Framework

Agentic AI Lens



Agentic AI Lens: AWS Well-Architected Framework

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Abstract and introduction	i
What makes this lens unique	1
Responsible agentic AI	2
The agentic AI technology landscape on AWS	2
About this lens	3
How to use this lens	4
Lens roadmap	4
Lens availability	5
Definitions	6
Design principles	12
Operational excellence	13
Design principles	13
Operational practices for agentic AI systems	14
Capability intent	14
Maturity levels	15
Common issues to watch for	17
AGENTOPS01-BP01 Establish well-defined agent roles, responsibilities, and success criteria	18
AGENTOPS01-BP02 Design multi-agent handoff procedures with human-in-the-loop escalation	22
AGENTOPS01-BP03 Develop test scenarios that accurately capture failures of dependent components, orchestration protocols, and business processes	25
Prompt and configuration lifecycle management	29
Capability intent	30
Maturity levels	30
Common issues to watch for	33
AGENTOPS02-BP01 Evolve agent prompts, tool calls, and configurations to reflect evolving business needs	34
AGENTOPS02-BP02 Implement configuration drift detection and remediation	38
AGENTOPS02-BP03 Implement agent behavior versioning and rollback capabilities	41
AGENTOPS02-BP04 Maintain feedback control loops for continuous improvement	45
Agent lifecycle and deployment processes	49
Capability intent	49
Maturity levels	50

Common issues to watch for	53
AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance	54
AGENTOPS03-BP02 Implement CI/CD pipelines tailored to agentic system deployment (AgentOps)	57
AGENTOPS03-BP03 Implement agent-specific scaling policies and capacity planning	61
AGENTOPS03-BP04 Implement organizational agent portfolio management and governance at scale	64
Tool integration and management practices	68
Capability intent	68
Maturity levels	69
Common issues to watch for	72
AGENTOPS04-BP01 Implement tool registry and catalog management	73
AGENTOPS04-BP02 Establish standardized tool integration protocols (MCP, A2A)	76
AGENTOPS04-BP03 Develop fallback behavior and error handling for tool invocations	79
Observability and monitoring for agentic systems	82
Capability intent	82
Maturity levels	83
Common issues to watch for	86
AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations	87
AGENTOPS05-BP02 Monitor agent behavior patterns and detect anomalies	90
AGENTOPS05-BP03 Implement structured logging and comprehensive audit trails	93
AGENTOPS05-BP04 Define and track KPIs for agent workflows	97
AGENTOPS05-BP05 Create workflow-specific dashboards for operational health	99
Testing, evaluation, and validation frameworks	102
Capability intent	102
Maturity levels	103
Common issues to watch for	106
AGENTOPS06-BP01 Design multi-layered testing frameworks	107
AGENTOPS06-BP02 Evaluate and track ongoing agent performance	110
AGENTOPS06-BP03 Establish SME-driven validation and business approval workflows	114
Operational recovery and consumption monitoring	117
Capability intent	117
Maturity levels	117
Common issues to watch for	120
AGENTOPS07-BP01 Implement automated response and recovery mechanisms	121

AGENTOPS07-BP02 Establish operational knowledge management systems	124
AGENTOPS07-BP03 Augment change management to accommodate technical improvements and business requirements	127
AGENTOPS07-BP04 Implement break-glass operational runbooks	130
Security	133
Design principles	133
Secure agent memory and state	134
Capability intent	134
Maturity levels	134
Common issues to watch for	137
AGENTSEC01-BP01 Implement memory isolation and integrity controls	138
AGENTSEC01-BP02 Validate and sanitize memory inputs	141
AGENTSEC01-BP03 Monitor for hallucination propagation	145
Secure agent tool usage	149
Capability intent	149
Maturity levels	150
Common issues to watch for	153
AGENTSEC02-BP01 Implement tool authorization	154
AGENTSEC02-BP02 Validate tool inputs and outputs	158
AGENTSEC02-BP03 Maintain approved tool registry with security assessments	161
Agent identity and permission management	166
Capability intent	167
Maturity levels	167
Common issues to watch for	171
AGENTSEC03-BP01 Implement strong authentication for agent identities	172
AGENTSEC03-BP02 Separate agent and human user permission	175
AGENTSEC03-BP03 Implement least privilege with dynamic boundaries	180
AGENTSEC03-BP04 Regular permission audits and access reviews	184
Agent goal alignment and manipulation prevention	187
Capability intent	188
Maturity levels	188
Common issues to watch for	191
AGENTSEC04-BP01 Implement guardrails and alignment controls	192
AGENTSEC04-BP02 Human-in-the-loop for critical decisions	196
Agent observability and non-repudiation	200
Capability intent	201

Maturity levels	201
Common issues to watch for	205
AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage	206
AGENTSEC05-BP02 Implement distributed tracing for agent interactions	212
Secure multi-agent orchestration	215
Capability intent	215
Maturity levels	216
Common issues to watch for	219
AGENTSEC06-BP01 Encrypt and sign inter-agent messages	220
AGENTSEC06-BP02 Implement workflow orchestration security controls	223
AGENTSEC06-BP03 Establish trust boundaries between agents	227
AGENTSEC06-BP04 Monitor and detect coordination anomalies	230
Human oversight protection and agent containment	233
Capability intent	234
Maturity levels	234
Common issues to watch for	238
AGENTSEC07-BP01 Implement cognitive load management	239
AGENTSEC07-BP02 Clear confidence indicators and manipulation warnings	242
AGENTSEC07-BP03 Multiple reviewers for critical operations	246
AGENTSEC07-BP04 Behavioral anomaly detection and agent containment	248
AGENTSEC07-BP05 Regular security assessments and red teaming	252
Secure agent inputs and outputs	255
Capability intent	256
Maturity levels	256
Common issues to watch for	260
AGENTSEC08-BP01 Multi-layer input validation and prompt injection defense	261
AGENTSEC08-BP02 Output filtering for sensitive information	265
Agent vulnerability scanning and penetration testing	268
Capability intent	269
Maturity levels	269
Common issues to watch for	273
AGENTSEC09-BP01 Integrate AI-powered vulnerability scanning across the development lifecycle	274
AGENTSEC09-BP02 Conduct context-aware penetration testing with multi-agent attack simulation	277

AGENTSEC09-BP03 Implement continuous security validation with automated remediation	280
AGENTSEC09-BP04 Establish scoped and controlled testing environments for agent security assessments	283
AGENTSEC09-BP05 Implement runtime threat detection, security event correlation, and automated remediation for agents	286
Reliability	291
Design principles	291
Predictable agent behavior	292
Capability intent	134
Maturity levels	134
Common issues to watch for	137
AGENTREL01-BP01 Implement a resilient messaging layer	297
AGENTREL01-BP02 Establish modular, fault-isolated layers	300
AGENTREL01-BP03 Design specialized agents following actor model principles	302
AGENTREL01-BP04 Standardize communication protocols	305
AGENTREL01-BP05 Implement adaptive provisioning	307
Predictable task execution	310
Capability intent	149
Maturity levels	150
Common issues to watch for	153
AGENTREL02-BP01 Design agents for specific and atomic tasks	315
AGENTREL02-BP02 Limit agent permissions to minimum required access	318
AGENTREL02-BP03 Implement behavioral anomaly detection and monitoring	320
AGENTREL02-BP04 Develop clear instruction protocols for agents	323
AGENTREL02-BP05 Establish tiered human oversight and approval workflows	325
Agent memory and state management	328
Capability intent	167
Maturity levels	167
Common issues to watch for	171
AGENTREL03-BP01 Design an information classification model to identify short-term and long-term memories	333
AGENTREL03-BP02 Architect fault-tolerant memory stores with redundancy and failover	336
AGENTREL03-BP03 Implement comprehensive state management and checkpoint-based recovery	338

AGENTREL03-BP04 Implement graceful degradation for memory and state operations	341
Multi-agent orchestration	344
Capability intent	188
Maturity levels	188
Common issues to watch for	191
AGENTREL04-BP01 Implement the arbiter agent pattern for coordinated multi-agent systems	348
AGENTREL04-BP02 Classify agents with a thorough capability taxonomy	351
AGENTREL04-BP03 Implement fallback mechanisms and graceful degradation for collaborative workflows	353
AGENTREL04-BP04 Implement resilient control planes for agent coordination	356
Agent cognition	359
Capability intent	201
Maturity levels	201
Common issues to watch for	205
AGENTREL05-BP01 Design modular, fault-tolerant agentic reasoning components	363
AGENTREL05-BP02 Facilitate reliable adaptation through evaluation-driven improvement cycles	366
AGENTREL05-BP03 Ground agent cognition in real information	368
Legacy system integration	371
Capability intent	215
Maturity levels	216
Common issues to watch for	219
AGENTREL06-BP01 Develop agent-based integrations with existing or legacy systems	377
AGENTREL06-BP02 Establish fallback mechanisms for legacy system degradation	380
AGENTREL06-BP03 Regularly test degraded system performance	382
AGENTREL06-BP04 Implement idempotent task execution patterns	385
AGENTREL06-BP05 Implement dynamic capability toggling	387
Agent monitoring, management and recovery	390
Capability intent	234
Maturity levels	234
Common issues to watch for	238
AGENTREL07-BP01 Design workflows in stages with incremental recovery	395
AGENTREL07-BP02 Enable automatic recovery from agent execution failures	397
AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery	400

Graceful degradation and configuration management	403
Capability intent	256
Maturity levels	256
Common issues to watch for	260
AGENTREL08-BP01 Establish consistent configuration management practices	408
AGENTREL08-BP02 Implement agent tracing for telemetry throughout agent processing	410
AGENTREL08-BP03 Architect agent systems with resource isolation and contention mitigation	413
AGENTREL08-BP04 Track agent memory utilization metrics	416
Performance efficiency	419
Design principles	419
Strategic performance planning and measurement	420
Capability intent	134
Maturity levels	134
Common issues to watch for	137
AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads	426
AGENTPERF01-BP02 Implement comprehensive performance telemetry	431
AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets	436
Core processing and reasoning pipeline optimization	441
Capability intent	149
Maturity levels	150
Common issues to watch for	153
AGENTPERF02-BP01 Design efficient reasoning pipelines	446
AGENTPERF02-BP02 Implement task-appropriate model selection strategies	451
AGENTPERF02-BP03 Optimize agent execution paths for reduced latency	457
AGENTPERF02-BP04 Optimize streaming responses and time-to-first-token for agent interactions	462
Memory, context, and RAG optimization	468
Capability intent	167
Maturity levels	167
Common issues to watch for	171
AGENTPERF03-BP01 Implement tiered memory management systems	474
AGENTPERF03-BP02 Optimize context window utilization and prompt management	480
AGENTPERF03-BP03 Optimize RAG retrieval pipelines for latency and precision	485
AGENTPERF03-BP04 Establish efficient agent caching and data access patterns	490

AGENTPERF03-BP05 Implement agentic retrieval patterns for dynamic, agent-driven knowledge access	493
Communication and protocol efficiency	496
Capability intent	188
Maturity levels	188
Common issues to watch for	191
AGENTPERF04-BP01 Optimize asynchronous message handling patterns	500
AGENTPERF04-BP02 Implement efficient protocol-based agent communications	503
AGENTPERF04-BP03 Design high-performing event-driven integration patterns	506
Workflow orchestration and multi-agent collaboration	508
Capability intent	201
Maturity levels	201
Common issues to watch for	205
AGENTPERF05-BP01 Design efficient workflow orchestration patterns	514
AGENTPERF05-BP02 Implement optimized multi-agent collaboration models	518
AGENTPERF05-BP03 Optimize multi-stage AI pipeline execution	521
AGENTPERF05-BP04 Implement efficient agent delegation and handoff patterns	524
Tool integration and framework optimization	527
Capability intent	215
Maturity levels	216
Common issues to watch for	219
AGENTPERF06-BP01 Design optimized tool integration strategies	533
AGENTPERF06-BP02 Implement efficient tool invocation patterns	536
AGENTPERF06-BP03 Optimize meta-tool utilization and tool chaining	539
Multi-tenancy and resource optimization	542
Capability intent	234
Maturity levels	234
Common issues to watch for	238
AGENTPERF07-BP01 Design efficient multitenant agent deployment models	547
AGENTPERF07-BP02 Implement tenant-aware performance isolation and throttling	550
Cost optimization	553
Design principles	553
Reasoning and execution cost optimization	554
Capability intent	134
Maturity levels	134
Common issues to watch for	137

AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops	559
AGENTCOST01-BP02 Optimize multi-agent collaboration cost through efficient handoff patterns	562
AGENTCOST01-BP03 Implement cost-effective patterns like hybrid supervisor for multi-agent coordination	566
AGENTCOST01-BP04 Design agent hierarchies and delegation patterns that reduce coordination overhead	569
Model invocation and token cost optimization	572
Capability intent	149
Maturity levels	150
Common issues to watch for	153
AGENTCOST02-BP01 Architect tiered model selection for cost-performance optimization	577
AGENTCOST02-BP02 Cost optimize token consumption through efficient prompt engineering	581
AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations	585
AGENTCOST02-BP04 Implement model customization for long-term cost reduction	588
Agent memory and state cost management	591
Capability intent	167
Maturity levels	167
Common issues to watch for	171
AGENTCOST03-BP01 Design cost-effective retrieval systems with tiered memory	595
AGENTCOST03-BP02 Cost optimize through intelligent compression and pruning of context windows	599
AGENTCOST03-BP03 Implement cost-optimized state persistence and lifecycle management	602
Agent tool serving cost optimization	605
Capability intent	188
Maturity levels	188
Common issues to watch for	191
AGENTCOST04-BP01 Design cost effective tool selection to minimize unnecessary invocations	610
AGENTCOST04-BP02 Cost optimize tool serving through serverless and resource sharing	613
AGENTCOST04-BP03 Implement intelligent caching and failure handling for tool results ..	617

Agent cost visibility and attribution	620
Capability intent	201
Maturity levels	201
Common issues to watch for	205
AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution	625
AGENTCOST05-BP02 Implement distributed cost tracing for multi-agent workflows	629
AGENTCOST05-BP03 Design tenant-aware cost allocation for agent as a service (AaaS) pricing models	632
AGENTCOST05-BP04 Create chargeback and ROI reporting	635
Agent discovery and deployment cost optimization	639
Capability intent	215
Maturity levels	216
Common issues to watch for	219
AGENTCOST06-BP01 Implement lightweight discovery and registry for cost-effective collaboration	644
AGENTCOST06-BP02 Cost optimize versioning and deployment through efficient artifact management	647
AGENTCOST06-BP03 Design cost-efficient initialization through warm pools and caching	650
Agent cost governance and continuous optimization	653
Capability intent	234
Maturity levels	234
Common issues to watch for	238
AGENTCOST07-BP01 Implement automated cost controls with intelligent cutoffs	659
AGENTCOST07-BP02 Establish proactive anomaly detection for agent cost patterns	663
AGENTCOST07-BP03 Create systematic optimization feedback loops for continuous improvement	666
Sustainability	671
Design principles	671
Resource reusability	672
Capability intent	134
Maturity levels	134
Common issues to watch for	137
AGENTSUS01-BP01 Design specialized agents with explicit resource boundaries	678
AGENTSUS01-BP02 Implement reusable workflow patterns	682
AGENTSUS01-BP03 Optimize resource utilization through shared services	686

AGENTSUS01-BP04 Scale cognitive processing pathways appropriately	689
AGENTSUS01-BP05 Adopt specification-driven tasks for frontier agents and long-running workflows	692
Resource right-sizing	696
Capability intent	149
Maturity levels	150
Common issues to watch for	153
AGENTSUS02-BP01 Optimize context management and memory utilization	701
AGENTSUS02-BP02 Establish efficient agent caching strategies	704
AGENTSUS02-BP03 Appropriately scale data, networking, and compute dependencies	707
AGENTSUS02-BP04 Measure and optimize the environmental footprint of agent workloads	710
Agent governance	713
Capability intent	167
Maturity levels	167
Common issues to watch for	171
AGENTSUS03-BP01 Maintain organizational skills and competencies	718
AGENTSUS03-BP02 Build agents to mirror your organizational skills and competencies ...	721
AGENTSUS03-BP03 Maintain comprehensive specifications for agents and agentic systems	724
AGENTSUS03-BP04 Decommission unused agents and prevent agent sprawl	727
Additional resources	731
Conclusion	732
Key themes	732
The evolving field	733
Next steps	733
Contributors	734
Acknowledgments	735
Appendix A: Best practice reference	736
Operational excellence	736
AGENTOPS01: How do you establish operational practices for agentic AI systems?	736
AGENTOPS02: How do you manage prompt and configuration lifecycle?	736
AGENTOPS03: How do you manage agent lifecycle and deployment processes?	737
AGENTOPS04: How do you establish tool integration and management practices?	738
AGENTOPS05: How do you implement comprehensive observability and monitoring for agentic systems?	738

AGENTOPS06: How do you implement testing, evaluation, and validation frameworks?	739
AGENTOPS07: How do you establish operational recovery and consumption monitoring?	739
Security	740
AGENTSEC01: How do you secure agentic memory and securely manage state between agents?	740
AGENTSEC02: How do you control and secure agent tool usage?	740
AGENTSEC03: How do you manage agent identities, permissions, and prevent privilege escalation?	741
AGENTSEC04: How do you support agent goal alignment and prevent manipulation?	741
AGENTSEC05: How do you implement observability and prevent repudiation?	742
AGENTSEC06: How do you secure multi-agent orchestration and coordination?	742
AGENTSEC07: How do you protect human oversight from manipulation and detect rogue agents?	743
AGENTSEC08: How do you validate and secure agent inputs and outputs?	743
AGENTSEC09: How do you perform vulnerability scanning and penetration testing for agentic AI systems?	744
Reliability	744
AGENTREL01: How do I develop reliable agentic systems?	744
AGENTREL02: How do you develop agentic systems that reliably execute tasks with predictable outcomes?	745
AGENTREL03: How do you support agent memory and state remaining reliably accessible throughout the agent lifecycle?	746
AGENTREL04: How do you orchestrate multi-agent systems to reliably execute tasks?	746
AGENTREL05: How do you implement reliable agent cognition that accesses the right data at the right time?	747
AGENTREL06: How do agents integrate effectively with existing systems without impacting the reliability of established processes?	747
AGENTREL07: How do fault tolerant agent systems recover?	748
AGENTREL08: How do agents determine when and where graceful degradation is appropriate?	748
Performance efficiency	749
AGENTPERF01: How do you plan strategically for agent performance and establish measurement practices?	749
AGENTPERF02: How do you optimize core agent processing and cognitive pipelines?	750

AGENTPERF03: How do you optimize memory management, context windows, and retrieval-augmented generation?	750
AGENTPERF04: How do you achieve efficient communication and protocol usage across agent interactions?	751
AGENTPERF05: How do you optimize workflow orchestration and multi-agent collaboration for performance?	751
AGENTPERF06: How do you optimize tool integrations and framework usage for agent performance?	752
AGENTPERF07: How do you manage multitenant performance isolation and optimize resource utilization?	752
Cost optimization	753
AGENTCOST01: How do you optimize agent reasoning and execution costs?	753
AGENTCOST02: How do you optimize agent model invocation and token consumption costs?	753
AGENTCOST03: How do you manage agent memory and state costs efficiently?	754
AGENTCOST04: How do you optimize agent tool invocation?	754
AGENTCOST05: How do you implement cost attribution?	755
AGENTCOST06: How do you optimize agent discovery registry and deployment costs?	756
AGENTCOST07: How do you establish agent cost governance and continuous optimization?	756
Sustainability	757
AGENTSUS01: How do you build sustainable and repeatable frameworks for managing compute, memory, and other shareable agent resources?	757
AGENTSUS02: How do I establish sustainable frameworks for agent dependencies?	757
AGENTSUS03: How do I establish durable patterns for agent interactions with users and business processes?	758
Summary	759
Document revisions	760
Notices	761
AWS Glossary	762

Agentic AI Lens - AWS Well-Architected

Publication date: **June 10, 2026** ([Document revisions](#))

Organizations deploying agentic AI are moving from asking "can we build an agent?" to "can we run agents reliably, securely, and cost-effectively at scale?" The AWS Well-Architected Agentic AI Lens answers this question by extending the AWS Well-Architected Framework with best practices specific to the design, deployment, and operation of agentic AI systems on AWS, providing architectural guidance from agent prototypes to production-grade agent systems.

This lens covers the full spectrum of agentic AI architecture, from the foundational infrastructure that supports agent compute and memory, to the orchestration patterns that coordinate multi-agent workflows, to the operational practices that help agents remain reliable, secure, and cost-effective in production. Whether agents serve end users through conversational interfaces, automate internal workflows, or operate on production infrastructure alongside human engineers, the architectural principles in this lens apply across the full range of agentic use cases.

What makes this lens unique

Agentic AI systems introduce architectural dimensions that existing cloud and generative AI guidance doesn't address:

- **Agents reason, not only respond:** A single user request might trigger multiple LLM inference calls, tool invocations, memory retrievals, and inter-agent communications, each adding latency, cost, and failure surface. Traditional request-response optimization doesn't address iterative reasoning loops.
- **Agents act autonomously:** Agents invoke tools, modify data, and interact with external systems without explicit human instruction at each step. This autonomy demands security controls, permission boundaries, and human oversight patterns purpose-built for autonomous operations.
- **Agent behavior is stochastic:** LLM-powered decisions are inherently non-deterministic, the same input might produce different outputs across invocations. Reliability strategies must account for this through behavioral monitoring, evaluation frameworks, and graceful degradation rather than deterministic testing alone.
- **Agents collaborate:** Multi-agent systems introduce coordination overhead, handoff complexity, and distributed failure modes. Orchestration patterns, communication protocols, and conflict resolution mechanisms become first-class architectural concerns.

- **Agents remember:** Persistent memory across sessions enables personalization and learning, but introduces data integrity, privacy, and cost management challenges that stateless applications don't face.

Responsible agentic AI

Agents that take actions in the real world carry greater responsibility than systems that only generate text. This lens embeds responsible AI principles throughout its best practices rather than treating them as a separate concern:

- **Bounded autonomy:** Every agent operates within explicitly defined scope boundaries, with guardrails that constrain behavior regardless of inputs received (see [AGENTSEC04](#)).
- **Transparency and explainability:** Agent decisions are logged, traced, and auditable, enabling teams to reconstruct exactly what happened during any execution (see [AGENTOPS05](#)).
- **Human oversight:** Tiered oversight models match the level of human review to the risk and reversibility of each agent action, preserving human judgment where it matters most (see [AGENTRELO2-BP05](#)).
- **Goal alignment:** Evaluation frameworks continually assess whether agents achieve intended objectives rather than pursuing misaligned goals (see [AGENTOPS06](#)).
- **Organizational sustainability:** Agent adoption preserves critical human expertise and institutional knowledge rather than creating dependencies on systems that only original developers understand (see [AGENTSUS03](#)).

The agentic AI technology landscape on AWS

This lens is built around the AWS services and open source frameworks that form the foundation for production agentic AI:

- **Amazon Bedrock** provides access to foundation models from multiple providers, Anthropic, OpenAI, Meta, Amazon, Mistral, NVIDIA, and others, with built-in capabilities for guardrails, knowledge bases, prompt management, and model evaluation.
- **Amazon Bedrock AgentCore** provides purpose-built infrastructure for deploying and operating agents at scale. AgentCore is framework-agnostic and model-agnostic. It is designed to support agentic frameworks such as Strands Agents, LangGraph, CrewAI, or your own, alongside major model providers. Its capabilities include Runtime for serverless agent hosting, Gateway for MCP-

compatible tool integration, Memory for managed agent state, Identity for agent authentication, Observability for distributed tracing, Evaluations for quality assessment, and Policy for Cedar-based access control.

- **Strands Agents** is an open source, model-driven framework for building agents with native support for MCP tools, A2A protocol, and multi-agent patterns (Graph, Swarm, and Workflow). Strands supports multiple model providers including Amazon Bedrock, Anthropic, OpenAI, Ollama, and others.
- **MCP and A2A** are open protocols for agent-to-tool and agent-to-agent communication that enable interoperability across frameworks, platforms, and the broader agent ecosystem.
- **Kiro** is an agentic IDE that accelerates agent development through spec-driven workflows, steering files for team standards, and hooks for automated quality checks.

About this lens

This lens is organized around the six pillars of the AWS Well-Architected Framework, each adapted to address the unique characteristics of agentic AI systems:

- **Operational excellence:** Running and improving autonomous agent systems through systematic operational practices, including prompt lifecycle management, behavioral monitoring, and human-in-the-loop governance.
- **Security:** Securing agent identities, tool access, and data flows while protecting against prompt injection, privilege escalation, and manipulation of autonomous operations.
- **Reliability:** Building agent systems that execute tasks predictably, recover from failures automatically, and maintain partial functionality under adverse conditions.
- **Performance efficiency:** Optimizing cognitive pipelines, model selection, memory access, and multi-agent coordination to deliver responsive and scalable agent performance.
- **Cost optimization:** Designing agents with cost as a primary consideration, right-sizing model and memory capabilities, and implementing full cost visibility.
- **Sustainability:** Building agent systems with modular, reusable architectures that maximize resource efficiency and maintain sustainable adoption patterns.

How to use this lens

This lens is intended to be used alongside the [AWS Well-Architected Framework](#). Each pillar in this lens contains focus areas with guiding questions, best practices, implementation guidance, and code examples specific to agentic AI workloads.

Use this lens when:

- Designing a new agentic AI system or adding agentic capabilities to an existing application
- Reviewing an existing agentic AI deployment for architectural improvements
- Evaluating the reliability, security, or cost profile of an agent-based workflow
- Establishing organizational standards for agentic AI development and operations

Lens roadmap

Not every team needs every best practice. Use these reading paths based on where you are in your agentic AI journey:

Building your first agent

Start with the foundational practices that establish agent scope, security boundaries, and predictable behavior:

- [AGENTOPS01: Define agent roles, success criteria, and handoff procedures](#)
- [AGENTREL02: Design for atomic tasks, least-privilege permissions, and clear instruction protocols](#)
- [AGENTSEC03: Establish agent identity, least-privilege access, and strong authentication](#)
- [AGENTSEC08: Validate inputs and filter outputs before going to production](#)

Moving to production

Add observability, evaluation, cost controls, and cognitive pipeline optimization:

- [AGENTOPS05: Implement tracing, anomaly detection, and operational dashboards](#)
- [AGENTOPS06: Establish testing and evaluation frameworks with LLM-as-judge](#)
- [AGENTPERF02: Optimize cognitive pipelines, model selection, and execution paths](#)
- [AGENTCOST01: Control reasoning loop costs and multi-agent coordination overhead](#)

- [AGENTCOST02: Right-size model selection and optimize token consumption](#)

Scaling to multi-agent systems

Address orchestration reliability, coordination security, and workflow performance:

- [AGENTREL04: Implement arbiter patterns, capability taxonomies, and fallback mechanisms](#)
- [AGENTPERF05: Optimize orchestration patterns and multi-agent collaboration models](#)
- [AGENTSEC06: Secure inter-agent communication, trust boundaries, and coordination](#)
- [AGENTCOST05: Implement cost attribution across multi-agent workflows](#)

Hardening an existing deployment

Strengthen security controls, resilience, and human oversight:

- [AGENTSEC04: Implement guardrails and human-in-the-loop for critical decisions](#)
- [AGENTSEC07: Protect human oversight and detect rogue agents](#)
- [AGENTREL06: Integrate with legacy systems reliably with fallbacks and idempotency](#)

Lens availability

Custom lenses extend the best practice guidance provided by AWS Well-Architected Tool. AWS WA Tool allows you to create your own [custom lenses](#), or to use lenses created by others that have been shared with you.

To begin reviewing your agentic AI workload, download and import the [Agentic AI Lens](#) into AWS WA Tool from the public [AWS Well-Architected custom lens GitHub repository](#).

Definitions

This section defines key terms used throughout the AWS Well-Architected Agentic AI Lens. These definitions establish a shared vocabulary for discussing agentic AI architecture and are used consistently across all pillars and best practices.

Agent

An autonomous software system that uses a large language model (LLM) as its reasoning engine to perceive context, plan actions, execute tasks, and adapt its behavior in pursuit of a defined goal. Agents differ from conventional AI applications in their ability to take multi-step actions, invoke external tools, maintain state across interactions, and make decisions without explicit human instruction at each step.

Agentic AI system

A software system composed of one or more agents that collaborate to accomplish complex, multi-step tasks. Agentic AI systems might include single agents with multiple tools, multi-agent architectures where specialized agents collaborate under orchestration, or hybrid systems that combine agentic and conventional software components.

Agentic RAG

A retrieval-augmented generation pattern where the agent actively controls the retrieval process as part of its reasoning loop, deciding when to retrieve, what to retrieve, which retrieval tool to use, and whether the retrieved context is sufficient before proceeding. Unlike standard RAG where retrieval is a static preprocessing step, agentic RAG treats retrieval as a dynamic reasoning action that the agent invokes iteratively based on its evolving understanding of the task. Agentic RAG patterns include iterative retrieval, query decomposition, retrieval sufficiency evaluation, and tool-augmented retrieval.

Agent cognition

The reasoning process by which an agent perceives its context, formulates a plan, selects actions, and evaluates outcomes. Agent cognition is powered by LLM inference and encompasses context retrieval, prompt construction, model inference, output parsing, and action selection.

Agent memory

The mechanisms by which agents store and retrieve information across interactions. Agent memory is typically organized into tiers:

- **Short-term (session) memory:** Transient context scoped to the current task or conversation, typically stored in fast in-memory stores with TTL-based expiration.
- **Long-term (persistent) memory:** Durable knowledge that persists across sessions, including learned preferences, historical outcomes, and domain knowledge.
- **Episodic memory:** Records of specific past interactions that can be retrieved to inform current decisions.
- **Semantic memory:** General domain knowledge stored as vector embeddings for semantic retrieval.
- **Procedural memory:** Learned patterns for task execution that inform how agents approach recurring task types.

Agent orchestration

The coordination of multiple agents or agent components to accomplish a complex task. Orchestration might be implemented by a dedicated orchestrator agent, a workflow engine (such as AWS Step Functions), or a combination of both. Orchestration patterns range from static workflows (fully defined at design time) to dynamic graph orchestration (where the agent's reasoning determines the next step at runtime).

Agent portfolio

The complete set of agents deployed across an organization, managed as organizational assets with tracked ownership, capabilities, dependencies, lifecycle state, cost profiles, and business value. Agent portfolio management enables organizations to help prevent agent sprawl, identify redundancy, manage cross-team dependencies, and make informed decisions about investment, consolidation, and retirement.

Arbiter agent

A specialized orchestration agent that resolves conflicts between specialized agents, manages resource contention, and helps keep the collective behavior of a multi-agent system aligned with the overall task objective.

Atomic task

A discrete, well-defined unit of work that an agent can execute independently with clear inputs, outputs, and success criteria. Atomic tasks are the building blocks of modular agent architectures.

Automatic cutoff (fail-fast)

A reliability pattern that detects when a downstream component is experiencing elevated error rates or latency and temporarily stops requests to that component, allowing it to recover without being overwhelmed. Automatic cutoffs operate in three modes: normal (requests flow), blocked (requests halted after failure threshold), and probing (periodic test requests to detect recovery). Also known as the circuit breaker pattern.

Capability taxonomy

A structured classification of agent capabilities that documents each agent's skills, limitations, resource requirements, and operational constraints. Capability taxonomies enable orchestration systems to make informed routing decisions and implement fallback strategies.

Context window

The maximum amount of text (measured in tokens) that a large language model can process in a single inference call. Context window utilization is a key performance and reliability metric for agent systems, as exceeding the context window causes inference failures.

Dead-letter queue (DLQ)

A message queue that captures messages that fail repeated processing attempts, helping prevent poison-pill messages from blocking healthy workflow execution. DLQs are a critical component of resilient agent messaging architectures.

Graceful degradation

The ability of an agent system to maintain partial functionality when components fail or performance degrades, rather than experiencing complete system failure. Graceful degradation involves detecting degraded conditions, activating fallback behaviors, and communicating reduced capabilities to users.

Guardrail

A control mechanism that constrains agent behavior within defined boundaries. Guardrails might be implemented at the model level (Amazon Bedrock Guardrails), the application level (input/output validation), or the infrastructure level (IAM policies, and network controls).

Hallucination

A phenomenon where a large language model generates plausible-sounding but factually incorrect or fabricated information. Hallucination is a key reliability risk in agentic AI systems, particularly when agents make decisions based on model-generated information rather than retrieved facts.

Human-in-the-loop (HITL)

An operational pattern where human judgment is incorporated into agent workflows at defined decision points. HITL workflows range from notification-only (humans are informed but don't intervene) to approval-required (agents can't proceed without explicit human authorization).

Idempotency

The property of an operation that produces the same result when executed multiple times with the same input. Idempotent operations can be safely retried after failures without risk of duplicate side effects.

Large language model (LLM)

A deep learning model trained on large text corpora that can generate, summarize, translate, and reason about text. LLMs serve as the reasoning engines for AI agents, enabling them to interpret instructions, plan actions, and generate responses.

Model Context Protocol (MCP)

An open standard protocol for connecting AI agents to external tools, data sources, and services. MCP defines a standardized interface for tool discovery, invocation, and result handling that enables agents to interact with a wide range of tools without custom integration code.

Multi-agent system

An architecture in which multiple specialized agents collaborate to accomplish tasks that exceed the capabilities of any single agent. Multi-agent systems might use hierarchical orchestration (supervisor-worker), peer-to-peer coordination, swarm-based collaboration, pipeline processing, dynamic graph orchestration, or hybrid patterns that combine multiple models within a single workflow.

Orchestrator agent

An agent responsible for decomposing complex tasks into subtasks, delegating subtasks to specialized agents, coordinating the flow of information between agents, and synthesizing results into a final output.

Prompt injection

A class of issue in which content in an agent's input attempts to override the agent's system prompt or cause it to perform actions outside its defined scope. Prompt injection is a key security consideration in agentic AI systems that process untrusted input.

Retrieval-augmented generation (RAG)

A technique that enhances LLM outputs by retrieving relevant information from external knowledge sources and including it in the model's context. RAG reduces hallucination rates and improves factual accuracy by grounding model reasoning in retrieved real-world information.

Saga pattern

A distributed transaction pattern that coordinates a sequence of local transactions across multiple services, with compensating transactions defined for each step to enable rollback when downstream steps fail. The saga pattern is commonly used in multi-agent workflows to maintain data consistency across agent boundaries.

Specialized agent

An agent designed to perform a single, well-defined capability with atomic operations. Specialized agents are the building blocks of multi-agent systems, each responsible for one discrete task within the larger workflow.

Stochasticity

The inherent randomness in LLM outputs that causes the same input to produce different outputs across invocations. Stochasticity is a fundamental characteristic of LLM-powered agents that must be accounted for in reliability and testing strategies.

Swarm

A multi-agent collaboration pattern where multiple agents work independently on overlapping aspects of a problem, sharing intermediate findings through a common workspace, without centralized supervision. Swarm patterns are effective for tasks that benefit from parallel exploration of a solution space, diverse perspectives, or emergent problem-solving. Swarm implementations require controls for redundant work prevention, convergence detection, and resource budgets to help prevent unbounded consumption.

Time-to-first-token (TTFT)

The elapsed time between a user submitting a request and receiving the first output token from the agent. TTFT is the primary perceived-performance metric for interactive agent interactions, as users perceive agents with low TTFT as dramatically faster even when total processing time is identical. TTFT is composed of pre-stream latency (input processing, context retrieval, and prompt construction) and inference TTFT (time from model invocation to first generated token).

Token

The basic unit of text processed by large language models. Tokens roughly correspond to word fragments, with approximately 750 words equaling 1,000 tokens. Token consumption is the primary cost driver for LLM-based agent systems.

Tool

A function or API that an agent can invoke to interact with external systems, retrieve information, or perform actions. Tools extend agent capabilities beyond text generation to include web search, database queries, code execution, API calls, and other operations.

Workflow

A structured sequence of agent actions, tool invocations, and decision points that accomplishes a complex task. Workflows might be implemented as Step Functions state machines, agent reasoning loops, or combinations of both.

Design principles

The Agentic AI Lens is built on a set of design principles that reflect the unique characteristics of autonomous AI agents. These principles guide architectural decisions across all six pillars and provide a foundation for evaluating agentic AI system designs.

- **Decompose agentic workloads into specialized, bounded agents:** Single-purpose agents with declared scope, explicit limits, and clear authority are easier to evaluate, secure, scale, evolve, and replace than monolithic ones. Decomposition is the foundation that every pillar's controls depend on.
- **Make every agent action observable and traceable end-to-end:** Reasoning steps, tool calls, memory accesses, and inter-agent handoffs need a single trace fabric so the same telemetry can answer cost, latency, quality, security, and footprint questions. Observability of agentic behavior is qualitatively different from infrastructure monitoring.
- **Treat agent behavior as code:** Prompts, tool catalogs, role definitions, model selections, and policies are versioned artifacts that go through peer review, testing, staged rollout, and rollback alongside the rest of the codebase.
- **Pair autonomy with proportionate human oversight:** Define autonomy levels (observer, assistant, autonomous, orchestrator) and escalation paths up front. Human review depth scales with the consequence of the action, by design rather than by reaction.
- **Ground autonomous behavior in explicit contracts:** Schemas, registries, structured success criteria, and confidence signals replace implicit assumptions about what an agent will do. What used to live inside a prompt becomes an enforceable interface.

Operational excellence

Organizations that operate agents through systematic, evidence-based practices ship improvements faster, recover from issues sooner, and maintain stakeholder confidence as agent portfolios scale. Operational excellence in agentic AI extends traditional DevOps and AIOps with agent-specific capabilities including prompt lifecycle management, behavioral monitoring, multi-faceted evaluation, and human-in-the-loop governance. Through iterative refinement, thorough observability, and evidence-based decision-making, organizations can operate agents reliably at scale while continually improving their performance, cost efficiency, and alignment with business objectives.

Capabilities

- [Operational practices for agentic AI systems](#)
- [Prompt and configuration lifecycle management](#)
- [Agent lifecycle and deployment processes](#)
- [Tool integration and management practices](#)
- [Observability and monitoring for agentic systems](#)
- [Testing, evaluation, and validation frameworks](#)
- [Operational recovery and consumption monitoring](#)

Design principles

In addition to the lens-level design principles, the operational excellence best practices in this lens are represented by at least one of the following principles:

- **Define purpose, autonomy, and success criteria before deployment:** Role descriptions, scope boundaries, and measurable outcomes are prerequisites for shipping. They drive guardrails, monitoring thresholds, and escalation triggers downstream.
- **Promote agents through a lifecycle with explicit gates:** SME-driven validation, CI/CD designed for non-deterministic systems, and staged rollouts prevent prompt, tool, or model changes from regressing production silently.
- **Detect drift and remediate automatically where safe:** Configuration drift, behavioral anomalies, and tool failures get detected, contained, and recovered without human intervention for routine cases. Humans are reserved for exceptions.

- **Operate by KPIs that map to business outcomes:** Resolution rate, escalation rate, customer satisfaction, and task completion sit alongside infrastructure metrics with equal weight in dashboards and reviews.
- **Manage agents as a portfolio:** Registries, catalogs, ownership records, and decommissioning processes treat agents as a population that needs governance, not as one-off projects.
- **Codify and recycle operational knowledge:** Runbooks, decision artifacts, change records, and incident learnings become structured inputs to future agent improvement and human operator training, not tribal knowledge.

Operational practices for agentic AI systems

Agentic systems that are built on strong operational foundations (with clear roles, reliable handoffs, and tested failure recovery) earn stakeholder trust and scale confidently into production. These systems combine precedent and best practices from several architectural disciplines, including generative AI and multi-agent coordination. These practices can't be combined without careful consideration. They must be judiciously selected and refined based on the workload being built and the business problem being addressed.

AGENTOPS01: How do you establish operational practices for agentic AI systems?

Capability intent

- Each agent has a documented purpose, measurable success criteria, and autonomy boundaries that stakeholders can trace back to a specific business outcome.
- Multi-agent coordination flows through standardized handoff protocols that transfer context reliably and route work to a human reviewer when confidence, stakes, or capability thresholds are exceeded.
- Agent systems are validated against realistic failure modes across dependent components, orchestration protocols, and business processes before they reach production, and on every behavioral change after.
- Operational signals, failure test outcomes, and business metrics feed a continuous improvement loop that keeps agent behavior aligned with evolving business needs.

- Operational artifacts such as job descriptions, handoff runbooks, and failure test scenarios are treated as living documents that stay in sync with the agents they describe.

Maturity levels

These levels summarize what each stage of maturity looks like for operational practices for agentic AI systems as a whole.

Level	Name	What it looks like
1	Initial	Agent scope, handoffs, and failure testing are ad-hoc. Knowledge about what each agent is meant to do, and how agents interact, lives with individual builders. Out-of-scope requests produce unpredictable responses, handoffs lose context, and failure modes are typically first encountered during production incidents.
2	Emerging	Each agent has a documented job description covering purpose, stakeholders, and measurable success criteria. Basic handoff runbooks and human escalation paths cover the most common scenarios . An initial failure test suite covers the happy path and a handful of dependent component failures, and reviews happen on an informal cadence.

Level	Name	What it looks like
3	Defined	Agent job descriptions, handoff protocols, and failure test suites are standardized across teams and stored in shared repositories. Amazon Bedrock Guardrails enforce scope at runtime, structured context packages accompany every handoff, and failure tests run in CI/CD as a mandatory gate across dependent components, orchestration protocols, and business processes. Handoff success rates and failure test pass rates are tracked as first-class metrics.
4	Proactive	Operational controls are automated and data-driven. AWS Agent Registry supports runtime discovery and intelligent routing across agents and tools. Amazon Bedrock AgentCore Evaluations run on a regular cadence to validate agent performance and resilience, and Amazon CloudWatch alarms on handoff and evaluation metrics trigger automated remediation.

Level	Name	What it looks like
5	Optimized	Operational practice is a continuously refined, self-improving system. Production incidents automatically generate new failure test scenarios, and agent job descriptions evolve through spec-driven development with reproducible audit trails. Multi-agent orchestration uses and contributes to industry standards such as Model Context Protocol and Agent-to-Agent communication . Resilience metrics and business outcome metrics are correlated in real time and feed directly into roadmap decisions.

Common issues to watch for

- Teams define agents by what they can do rather than what they are accountable for, leaving operators without a clear way to judge whether the agent is working or to decide when to escalate.
- Multi-agent workflows are assembled bottom-up from individual agent capabilities, without documented handoff protocols or human escalation paths, producing systems that are hard to operate when a single agent fails.
- Failure testing is treated as a phase rather than a living practice, so resilience degrades quietly with each prompt, tool, or model change that ships without a matching regression test.
- Success criteria and resilience metrics are tracked in silos separate from business metrics, so agent owners can't explain the business impact of a coordination failure or a drop in failure-test pass rate.

- Operational artifacts such as job descriptions, runbooks, and failure test scenarios are authored once and never revisited, so documented intent drifts away from actual runtime behavior.

Best practices

- [AGENTOPS01-BP01 Establish well-defined agent roles, responsibilities, and success criteria](#)
- [AGENTOPS01-BP02 Design multi-agent handoff procedures with human-in-the-loop escalation](#)
- [AGENTOPS01-BP03 Develop test scenarios that accurately capture failures of dependent components, orchestration protocols, and business processes](#)

AGENTOPS01-BP01 Establish well-defined agent roles, responsibilities, and success criteria

An agent without a documented role can be difficult to evaluate and improve. Provide each agent a clear purpose, scope, autonomy level, and success criteria to change ambiguous behavior into something teams can observe, measure, and hold accountable for.

Desired outcome:

- Every agent has a documented job description specifying role, owned business outcomes, autonomy boundaries, and measurable success criteria.
- Teams can objectively assess whether an agent performs as intended, and stakeholders understand what value each agent delivers.
- Failure handling and escalation procedures are defined before deployment so out-of-scope requests and edge cases are handled predictably.
- Success criteria map to business outcomes (task resolution rate, customer satisfaction) alongside technical metrics.

Common anti-patterns:

- Deploying agents without documented scope boundaries, producing unpredictable behavior when the agent encounters requests outside its intended purpose.
- Defining success criteria using only technical metrics (latency, uptime) without mapping to business outcomes, making it impossible to assess whether the agent delivers value.
- Treating agent role definitions as one-time artifacts rather than living documents that evolve with business requirements.

- Skipping the identification of stakeholders who depend on the agent, so there is no clear owner when behavior needs adjustment.

Benefits of establishing this best practice:

- Documented intent and scope become the foundation for downstream controls. Guardrails, monitoring thresholds, and escalation paths all derive from the agent's stated purpose.
- Measurable success criteria enable data-driven evaluation, giving the team empirical evidence for iterative refinement and investment decisions.
- Out-of-scope requests and edge cases are handled gracefully because failure paths are defined before the agent ships.
- Stakeholders and operators share a common understanding of what the agent is supposed to do and who is accountable for its behavior.

Level of risk exposed if this best practice is not established: High

Implementation guidance

A written role definition is the simplest control an agent can have, and the one most often skipped. Without it, guardrails get tuned against assumed behavior, monitoring thresholds get set against assumed workloads, and escalation triggers fire against assumed failures. The document should name the agent's primary purpose, the business process it supports, the stakeholders who depend on it, and the outcomes it is accountable for. Keep the document current as requirements evolve.

An agent that observes and reports is a different operational commitment from one that takes autonomous action, and conflating these two roles can become an oversight. The maturity progression from observer to assistant to autonomous to orchestrator to innovator gives stakeholders a common vocabulary for talking about how much agency an agent has and how much human review it still needs. Set the expectation on the right rung of that progression, and the downstream controls follow.

Success criteria fail when they measure what is cheap to measure rather than metrics that matter operationally. For example, a customer support agent with a latency target and no resolution rate is optimizing for the wrong thing. The SMART framework (specific, measurable, achievable, relevant, time-bound) helps, but the sharper test is to check if a metric improves alongside the business outcome it's measuring. Business outcome metrics (task resolution rate, escalation rate, customer satisfaction) should be checked alongside technical ones and share equal weight.

Operationalize the role definition by wiring it into runtime controls. [Amazon Bedrock Guardrails](#) turns documented topic restrictions, content filters, and denied topics into enforcement at invocation time. For no-code paths like [Amazon Quick Suite](#), the same role-definition discipline applies through identity, instructions, and knowledge configuration. Publish agent role definitions to [AWS Agent Registry](#) so that both human operators and other agents can discover capabilities, understand scope, and route work appropriately.

Failure handling specifies what the agent does when it encounters requests outside scope, when tools are unavailable, or when it can't produce a confident response. Graceful degradation paths, confidence-based escalation, and structured logging for out-of-scope requests keep edge cases from turning into incidents. Review job descriptions quarterly or whenever business requirements change.

Implementation steps

1. **Create an agent job description template:** Include name and identifier, primary purpose, stakeholder list, autonomy level on the maturity model, success criteria with measurable targets, out-of-scope topics, and escalation procedures.
2. **Complete the job description for each agent:** Engage technical and business stakeholders together to validate scope and success-criteria alignment, and capture sign-off from both.
3. **Define measurable success criteria:** Combine business outcome metrics (task completion rate, escalation rate, user satisfaction) with technical metrics, and apply the SMART framework to each.
4. **Enforce scope at runtime with [Amazon Bedrock Guardrails](#):** Configure topic restrictions, content filters, and denied topics that reflect the agent's documented boundaries.
5. **Publish agent definitions to [AWS Agent Registry](#):** Register each agent's capabilities, scope, and metadata so operators and other agents can discover and route work appropriately.
6. **Define failure handling and escalation:** Document graceful degradation paths, confidence-based human escalation triggers, and structured logging requirements for out-of-scope requests.
7. **Establish a quarterly review cadence:** Update job descriptions as business requirements change, and treat them as living operational artifacts owned by a named individual.

Resources

Related best practices:

- [AGENTOPS01-BP02 Design multi-agent handoff procedures with human-in-the-loop escalation](#)
- [AGENTOPS02-BP01 Evolve agent prompts, tool calls, and configurations to reflect evolving business needs](#)
- [AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance](#)
- [AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Agentic AI patterns and workflows on AWS](#)
- [AI agents in enterprises: Best practices with Amazon Bedrock AgentCore](#)
- [Kiro Specs](#)

Related videos:

- [AWS re:Invent 2024 - Agentic AI Meets responsible AI: Strategy and best practices \(AIM422\)](#)
- [AWS re:Invent 2024 - Agents in the enterprise: Best practices with AgentCore \(AIM3310\)](#)
- [AWS 2025 - Beginner-Friendly Amazon Bedrock AgentCore & Strands Agents Tutorial](#)
- [AWS re:Invent 2024 - Agentic AI and the journey to gen AI value realization \(AIM242\)](#)

Related examples:

- [GitHub: Amazon Bedrock Samples, GenAI Quick-Start PoCs](#)
- [GitHub: Sample Agentic Platform on AWS](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore](#)

Related services:

- [Amazon Bedrock Guardrails](#)
- [AWS Agent Registry](#)
- [Amazon Bedrock AgentCore](#)

- [Amazon CloudWatch](#)

AGENTOPS01-BP02 Design multi-agent handoff procedures with human-in-the-loop escalation

Without a structured context package, the receiving agent re-derives work the previous agent already finished. Lacking a defined escalation path means that high-stakes or low-confidence decisions can slip past human review.

Desired outcome:

- You have documented handoff protocols that transfer tasks between agents with full context and clear accountability.
- You have escalation paths that route requests to the right agent or human reviewer when an agent reaches its capability limits.
- You detect deadlocks and timeouts automatically and resolve them through documented recovery procedures.
- You monitor handoff latency, context transfer completeness, and collaboration success rates as first-class operational metrics.

Common anti-patterns:

- Implementing agent-to-agent handoffs without structured context packages, forcing the receiving agent to re-derive context and repeat work the delegating agent already completed.
- Relying solely on agent-to-agent escalation without defining agent-to-human triggers, leaving high-stakes or low-confidence decisions without human oversight.
- Deploying multi-agent workflows without deadlock detection or timeout handling, allowing circular dependencies between agents to stall the workflow indefinitely.
- Treating handoff failures as rare events, so no one tracks success rates or context-transfer completeness until a customer incident forces the investigation.

Benefits of establishing this best practice:

- Documented handoff runbooks and escalation procedures create repeatable collaboration patterns that reduce operational complexity.

- Tasks requiring human judgment reliably reach human reviewers without creating bottlenecks in routine agent-to-agent work.
- Context-rich handoffs help prevent duplicate reasoning, cutting both latency and token cost in multi-agent workflows.
- Deadlock detection and timeout handling keep transient coordination failures from becoming workflow-level outages.

Level of risk exposed if this best practice is not established: High

Implementation guidance

A handoff is a data contract before it is a workflow step. The sending agent must know what to include in the payload, while the receiving agent must know what to expect. When that contract is missing, each handoff becomes an improvised negotiation where the receiver either asks for more information (adding round trips) or guesses (adding errors). Standardized protocols such as Model Context Protocol (MCP) and agent-to-agent (A2A) communication give agents built on different frameworks a shared vocabulary for task description, completed work, memory artifacts, and handoff reason, so the contract stays stable across technology choices.

Discovery should be a part of the data contract. Agents need to know which peers exist, what they can do, and whether they are accepting work right now. [AWS Agent Registry](#) provides a centralized catalog that captures capabilities, availability, and metadata for agents and tools, making intelligent routing possible instead of hardcoded. [Amazon Bedrock AgentCore Gateway](#) then gives the workflow secure connectivity and tool invocation across agents, with every interaction auditable.

Escalation has two distinct triggers to consider:

1. Agent-to-agent escalation happens when a task needs capabilities outside the current agent's scope. This trigger is a routing decision.
2. Agent-to-human escalation happens when confidence drops below a threshold, the stakes are high, or the retry budget has been exhausted. This trigger is a judgment decision.

Mixing them together either sends too many routine tasks to humans (creating fatigue and bottlenecks) or sends too many high-stakes decisions through automated routing (creating risk). Dictate each trigger separately and verify that you can see when each one fires.

Deadlocks deserve their own attention because they are silent. Two agents can wait for each other indefinitely while every individual operation looks healthy.

A wait that exceeds a configurable timeout is a deadlock suspect, but the response has to be automated: task reassignment, human notification, or both. A deadlock that requires a human to notice is a deadlock that lasts until someone notices.

Implementation steps

1. **Document handoff runbooks:** Cover the top five agent-to-agent collaboration scenarios, specifying context package format, acceptance criteria, and expected outcomes.
2. **Define a structured context package schema:** Include task description, completed work, memory artifacts, and handoff reason. Version the schema so receivers can reject malformed handoffs.
3. **Deploy an agent registry:** Catalog agent capabilities, availability status, and handoff acceptance criteria in [AWS Agent Registry](#) to enable runtime discovery and intelligent routing.
4. **Connect agents through [Amazon Bedrock AgentCore Gateway](#):** Configure secure connectivity, tool invocation, and authorization across agents with auditable interaction records.
5. **Define escalation criteria:** Separate agent-to-agent triggers (capability mismatch) from agent-to-human triggers (confidence threshold, high-stakes decisions, retry budget exhaustion), and instrument each.
6. **Implement deadlock detection:** Configure alarms on workflow execution duration, and automate resolution through task reassignment and human notification.
7. **Monitor collaboration health:** Track success rates, handoff latency, and context-transfer completeness in [Amazon CloudWatch](#), with alerting on handoff failure rates.

Resources

Related best practices:

- [AGENTOPS01-BP01 Establish well-defined agent roles, responsibilities, and success criteria](#)
- [AGENTOPS04-BP01 Implement tool registry and catalog management](#)
- [AGENTOPS04-BP02 Establish standardized tool integration protocols \(MCP, A2A\)](#)
- [AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations](#)
- [AGENTPERF05-BP04 Implement efficient agent delegation and handoff patterns](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Agentic AI patterns and workflows on AWS](#)
- [Agentic AI frameworks, platforms, protocols, and tools on AWS](#)

Related videos:

- [AWS re:Invent 2024 - Amazon Bedrock Agents and AgentCore Design Patterns \(TNC322\)](#)
- [AWS re:Invent 2024 - Building Scalable, Self-Orchestrating AI Workflows with A2A and MCP \(DEV415\)](#)
- [AWS re:Invent 2024 - Anti-Money Laundering Multi-agent Orchestration with AWS Strands \(DEV326\)](#)

Related examples:

- [GitHub: Sample Multi-Agent SaaS Workshop](#)
- [GitHub: Sample Agentic Platform on AWS](#)
- [GitHub: Sample Bedrock AgentCore Multi-Tenant](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Agent Registry](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon CloudWatch](#)

AGENTOPS01-BP03 Develop test scenarios that accurately capture failures of dependent components, orchestration protocols, and business processes

Happy-path testing predicts how agents behave when everything works, while failure testing predicts how they behave in the event of unforeseen issues. A resilience posture built on injected

failures, deadlock scenarios, and disrupted business processes is the difference between graceful degradation and an unexpected outage.

Desired outcome:

- You have a failure test suite for every agent covering dependent-component failures, orchestration breakdowns, and business-process disruptions.
- You can inject failures into agent workflows on demand and verify that error handling, graceful degradation, and escalation behave as designed.
- You maintain known failure patterns as regression tests that run automatically on every behavioral change.
- You track failure test pass rates over time as a visible resilience metric.

Common anti-patterns:

- Testing only the happy path without validating agent behavior when tools are unavailable, APIs time out, or data sources return errors.
- Running failure tests only in isolated unit environments without simulating multi-agent coordination failures such as deadlocks, message loss, or handoff timeouts.
- Treating failure test scenarios as a one-time exercise rather than a living regression suite that grows with each production incident.
- Skipping tests for business-process disruptions, upstream format changes, delayed approvals, and downstream rejections, so agents fail silently when the real world shifts.

Benefits of establishing this best practice:

- Failure test suites become a stable benchmark for comparing agent resilience across iterations, providing the empirical basis for improvement decisions.
- Standardized testing helps assess every agent change for resilience impact the same way, regardless of who made the change or how urgent the timeline is.
- Known failure patterns captured as regression tests help prevent recurrence of previously diagnosed issues.
- Visible resilience trends give leadership and operators a shared view of whether the agent portfolio is getting more reliable or more fragile.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Failure modes cluster into three categories, and each requires a different testing posture.

Dependent-component failures are the most tractable. Tools return 5xx errors, APIs time out, knowledge bases return empty results, and model inference throttles or degrades. These map cleanly to agent evaluation and synthetic fault injection. [Amazon Bedrock AgentCore Evaluations](#) provides automated assessment of how agents handle edge cases and failure scenarios, using built-in and custom evaluators to score agent behavior against expected outcomes. For infrastructure-level fault injection such as network latency or capacity exhaustion, [AWS Fault Injection Service](#) complements agent evaluations by validating that retry policies, fallbacks, and cutoffs behave as documented at the infrastructure layer.

Orchestration breakdowns are harder because they require more than a single failing component. Message loss, duplicate delivery, out-of-order messages, deadlocks, handoff timeouts, and context-package corruption all emerge from the interactions between agents rather than any single agent's behavior. Test these scenarios by simulating coordination failures in your multi-agent workflows. Inject handoff timeouts, corrupt context packages, and trigger concurrent requests that expose race conditions. Decide whether to simulate these failures in a shared staging environment or in a dedicated chaos environment. The former catches regressions faster, while the latter reduces the scope of impact during exploratory testing.

Business-process disruptions are the category most often missed because they don't look like infrastructure failures. When an upstream team changes an input schema, when a required approval is delayed, or when a downstream system rejects an agent's output, the agent's code is intact and every dependency responds, but the workflow still fails. Test scenarios must cover how the agent behaves when the business process shifts, like graceful failure, meaningful error messages, appropriate escalation, and no silent corruption. These tests protect against the failure mode where the system looks healthy to monitoring but delivers wrong or incomplete outcomes.

Use [Amazon Bedrock AgentCore Evaluations](#) to maintain and run evaluation datasets that capture known failure patterns as regression tests. Integrate the evaluation suite into the CI/CD pipeline as a mandatory gate so deployments are blocked when failure-handling regression appears. Track pass rates in [Amazon CloudWatch](#) and configure alarms when resilience metrics degrade. The operational benefit compounds, as every production incident becomes an opportunity to add a test scenario, and the suite gets sharper over time.

Implementation steps

1. **Catalog known failure modes per agent:** Organize by dependent-component failures, orchestration breakdowns, and business-process disruptions, with a named owner for each category.
2. **Create tests for dependent-component failures:** Simulate tool unavailability, API timeouts, data-source errors, and model inference degradation using [Amazon Bedrock AgentCore Evaluations](#) dataset evaluations and custom evaluators.
3. **Create tests for orchestration breakdowns:** Cover communication failures, deadlock conditions, handoff errors, and context-package corruption by simulating coordination failures in multi-agent workflows.
4. **Create tests for business-process disruptions:** Simulate upstream process changes, input format changes, and downstream system rejections, and verify graceful failure and meaningful escalation.
5. **Integrate infrastructure fault injection where appropriate:** Use [AWS Fault Injection Service](#) for infrastructure-level failures (throttling, capacity exhaustion, network latency) that affect agent workflows.
6. **Gate deployments on failure-handling regression:** Make the evaluation suite a mandatory CI/CD stage that blocks promotion when resilience regressions appear.
7. **Maintain evaluation datasets as living regression suites:** Use [Amazon Bedrock AgentCore Evaluations](#) to track pass rates, and alert on degradation via [Amazon CloudWatch](#).
8. **Review and expand scenarios quarterly:** Incorporate failure patterns discovered in production incidents so the suite grows with the system.

Resources

Related best practices:

- [AGENTOPS02-BP01 Evolve agent prompts, tool calls, and configurations to reflect evolving business needs](#)
- [AGENTOPS04-BP03 Develop fallback behavior and error handling for tool invocations](#)
- [AGENTOPS06-BP01 Design multi-layered testing frameworks](#)
- [AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Evaluating AI agents: Real-world lessons from building agentic systems at Amazon](#)
- [Planning for failure: How to make generative AI workloads more resilient](#)
- [Guidance for Agentic AI Operational Foundations on AWS](#)

Related videos:

- [AWS re:Invent 2024 - Best practices for generative AI observability \(COP404\)](#)
- [AWS re:Invent 2024 - Unlock the power of generative AI with AWS Serverless \(SVS319\)](#)

Related examples:

- [GitHub: Open Source Bedrock Agent Evaluation](#)
- [GitHub: Sample Agentic Platform on AWS](#)

Related services:

- [Amazon Bedrock AgentCore Evaluations](#)
- [AWS Fault Injection Service](#)
- [Amazon CloudWatch](#)
- [AWS CodePipeline](#)

Prompt and configuration lifecycle management

Agent prompts, configurations, and tool permissions define agent behavior and evolve continually as business needs change. Managing these artifacts requires balancing rapid iteration with stability so that changes are tracked, tested, and reversible.

AGENTOPS02: How do you manage prompt and configuration lifecycle?

Capability intent

- Agent prompts, behavioral configurations, and tool permissions are managed as first-class operational artifacts with full version history, documented ownership, and a defined lifecycle from authoring through retirement.
- Behavioral changes reach production only after passing automated evaluation against documented quality thresholds, and they move through staged rollout rather than full-traffic deployment.
- Configurations remain consistent across development, staging, and production environments, and drift from approved baselines is detected and remediated automatically.
- Rollback to a known-good behavioral baseline is automated, tested regularly, and completes within minutes of the decision to revert.
- Operational signals, user feedback, and business outcome metrics flow into a structured improvement backlog that closes the loop from observation to validated change.

Maturity levels

These levels summarize what each stage of maturity looks like for prompt and configuration lifecycle management as a whole.

Level	Name	What it looks like
1	Initial	Prompts and behavioral configurations are embedded in application code or maintained by hand in each environment. Changes are deployed directly to production without evaluation, and there is no version history to roll back to when a change regresses. Configuration differences between development, staging, and

Level	Name	What it looks like
		production are common and only noticed after incidents.
2	Emerging	Prompts and behavioral configurations are stored outside application code in a versioned repository such as Amazon Bedrock Prompt Management . Infrastructure baselines are defined in AWS CloudFormation or the AWS CDK , and changes follow a documented lifecycle with peer review. Rollback is possible but manual.
3	Defined	A full prompt lifecycle (Draft, Review, Active, Archived) is enforced through AWS CodePipeline approval gates, with automated evaluation using Amazon Bedrock Evaluations as a promotion gate. AWS CloudFormation drift detection and AWS Config rules monitor for configuration drift, and AWS CloudTrail captures change attribution. Rollback runbooks are defined and tested.

Level	Name	What it looks like
4	Proactive	Behavioral changes roll out in stages using Amazon Bedrock AgentCore Runtime weighted routing, with automated promotion gates driven by Amazon CloudWatch quality metrics. A/B testing compares behavioral variants under traffic, and automated remediation restores approved configurations when drift is detected. A multi-channel feedback pipeline built on Amazon EventBridge feeds an improvement backlog.

Level	Name	What it looks like
5	Optimized	The full lifecycle is self-service for teams, with automated rollback that restores behavioral baselines within minutes of a quality threshold being exceeded. Cross-environment consistency is validated continuously, and Amazon Bedrock AgentCore Observability correlates deployment timestamps with quality trends so every metric change is attributable to a specific behavioral version. Feedback-to-improvement workflows close the loop from signal to validated change, and the organization contributes its patterns back to the broader agentic AI community.

Common issues to watch for

- Prompts are embedded directly in application code, so any behavioral change forces a full code deployment and makes it impossible to iterate on agent behavior independently.
- Configuration changes are applied through manual console edits in individual environments, which causes development, staging, and production to drift apart and produces "it worked in test" failures in production.
- Behavioral changes roll straight to 100 percent of traffic without staged rollout or pre-deployment evaluation, maximizing the scope of impact of any regression and leaving no safe path to catch issues early.

- Rollback is treated as a manual, exploratory activity rather than a tested, automated procedure, so recovery takes hours instead of minutes when a behavioral change goes wrong.
- User feedback, quality metrics, and business outcome signals are collected but never connected to specific prompt or configuration versions, leaving teams unable to attribute quality changes to the behavioral changes that caused them.

Best practices

- [AGENTOPS02-BP01 Evolve agent prompts, tool calls, and configurations to reflect evolving business needs](#)
- [AGENTOPS02-BP02 Implement configuration drift detection and remediation](#)
- [AGENTOPS02-BP03 Implement agent behavior versioning and rollback capabilities](#)
- [AGENTOPS02-BP04 Maintain feedback control loops for continuous improvement](#)

AGENTOPS02-BP01 Evolve agent prompts, tool calls, and configurations to reflect evolving business needs

A prompt shapes agent behavior more directly than almost any other configuration artifact. Create a prompt lifecycle that applies code-grade discipline, versioning, review, evaluation, and rollback to prompts, which helps avoid unnoticed prompt drift and degraded decisions in agent interactions.

Desired outcome:

- You manage agent prompts through a defined lifecycle: authoring, review, testing, deployment, monitoring, and retirement.
- Every production prompt has a documented version history, an evaluation record, and a clear owner.
- You deploy prompt updates independently of application code and roll back to a previous version within minutes.
- You track the performance impact of each prompt change over time and can attribute quality shifts to specific versions.

Common anti-patterns:

- Hardcoding prompts directly in application code, making it impossible to update agent behavior without a full code deployment and blocking independent prompt iteration.

- Deploying prompt changes directly to production without evaluation against quality benchmarks, discovering regressions only after users report them.
- Operating without prompt version history, making it impossible to determine which prompt change caused a behavioral regression or to roll back to a known-good version.
- Treating prompt changes as too small to require review, letting ad-hoc edits accumulate into drift that no one owns.

Benefits of establishing this best practice:

- Behavioral changes follow a consistent, auditable path from authoring to deployment, reducing operational risk and enabling reliable rollback.
- Prompt performance tracking and evaluation create an empirical basis for iteration. Each update is validated against measurable quality criteria before reaching production.
- Teams can deploy prompt changes independently of application code, shortening the feedback loop between business need and runtime behavior.
- Failed prompt updates revert in minutes rather than requiring an incident response.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Treat prompts as first-class operational artifacts with the same discipline applied to application code. Application code moves through version control, code review, automated tests, staged deployment, and documented rollback. Most organizations apply none of these to prompts, which can result in agent behavior drifts. Apply the same lifecycle to prompts as you do to code, where you name the stages explicitly and require changes to follow them.

[Amazon Bedrock Prompt Management](#) provides versioning, metadata, and integration with [Amazon Bedrock Evaluations](#). For no-code paths built with [Amazon Quick Suite](#), the same stages apply through the identity and instructions configuration. A four-stage lifecycle works for most teams:

1. Draft (under development, not deployed)
2. Review (under peer review and evaluation)
3. Active (deployed to production)
4. Archived (retired but retained for audit)

Gate stage transitions by approval workflows implemented in [AWS CodePipeline](#) or [AWS Step Functions](#), not by convention.

Every prompt entry needs required metadata: purpose, target agent, expected behavior, evaluation criteria, and owner. Parameterization should be used to reduce duplication across related agents. Steering files in Kiro or equivalent conventions codify these standards so they are applied automatically during development rather than enforced after the fact.

[Amazon Bedrock Evaluations](#) runs each prompt version against a standardized dataset and produces scores for task success, response relevance, and adherence to behavioral guidelines. A prompt that can't meet its minimum thresholds doesn't advance from review to active. Once in production, quality metrics published to [Amazon CloudWatch](#) as custom metrics give the team an early warning when a prompt that passed evaluation starts degrading in the real world, triggering a review workflow.

Implementation steps

1. **Stand up the central prompt repository:** Configure [Amazon Bedrock Prompt Management](#) with four lifecycle stages (draft, review, active, archived) and required metadata fields (purpose, target agent, expected behavior, evaluation criteria, owner).
2. **Define prompt authoring standards:** Specify required metadata, formatting conventions, and documentation requirements. Apply them through shared templates or steering files so they are enforced during development.
3. **Build versioned evaluation datasets:** Create datasets for each agent's primary use cases and store them with versioning enabled so evaluation results are reproducible.
4. **Gate transitions on automated evaluation:** Configure an [Amazon Bedrock Evaluations](#) pipeline that runs on every transition from draft to review, with minimum quality thresholds.
5. **Enforce lifecycle stages in CI/CD:** Use [AWS CodePipeline](#) to block promotion to active without approval and evaluation threshold checks.
6. **Reference prompts by ID, not by value:** Deploy agents with parameterized references to the prompt repository rather than hardcoded strings, so prompts evolve independently of application code.
7. **Monitor active prompts in production:** Publish quality metrics to [Amazon CloudWatch](#), build dashboards per agent, and configure alarms that trigger review workflows when thresholds are exceeded.

Resources

Related best practices:

- [AGENTOPS01-BP03 Develop test scenarios that accurately capture failures of dependent components, orchestration protocols, and business processes](#)
- [AGENTOPS02-BP03 Implement agent behavior versioning and rollback capabilities](#)
- [AGENTOPS06-BP01 Design multi-layered testing frameworks](#)
- [AGENTRELO2-BP04 Develop clear instruction protocols for agents](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Evolving software delivery for agentic AI](#)
- [AI agents in enterprises: Best practices with Amazon Bedrock AgentCore](#)
- [Kiro](#)
- [Kiro Steering](#)

Related videos:

- [AWS 2025 - Amazon Bedrock Prompt Management Demo](#)
- [AWS re:Invent 2024 - Responsible generative AI: Evaluation best practices and tools \(AIM342\)](#)

Related examples:

- [GitHub: Sample Bedrock Evaluation Adapter](#)
- [GitHub: Sample Bedrock Model Evaluation](#)
- [GitHub: Amazon Bedrock Samples, GenAI Quick-Start PoCs](#)

Related services:

- [Amazon Bedrock](#)
- [AWS CodePipeline](#)
- [Amazon S3](#)
- [Amazon CloudWatch](#)

AGENTOPS02-BP02 Implement configuration drift detection and remediation

Configurations can drift, creating outdated or unstable versions over time. For example, a manual tweak in one environment, a guardrail flag changed during an incident, or an experimental override never reverted can produce agents that behave differently in production than in testing. Automated drift detection catches these events before they turn into incidents.

Desired outcome:

- Agent configurations stay consistent with approved baselines across every environment.
- Unauthorized or unintended changes are detected and remediated automatically.
- Every configuration change follows a documented approval workflow with a full audit trail.
- Cross-environment consistency is validated continually so development, staging, and production don't drift apart.

Common anti-patterns:

- Managing agent configurations through manual console changes without version control, making it impossible to track what changed, when, and by whom.
- Allowing different environments to drift apart without automated consistency checks, so agents behave differently in production than in testing.
- Detecting configuration drift only after it causes a production incident rather than through proactive monitoring.
- Treating behavioral configurations (system prompts, guardrail settings) as low-risk and skipping approval workflows for changes that fundamentally alter agent behavior.

Benefits of establishing this best practice:

- Automated drift detection helps keep agent configurations inside approved boundaries continually, supporting audit requirements and reducing the risk of unauthorized behavioral change.
- Configuration monitoring provides visibility beyond runtime metrics, exposing issues at the configuration layer before they manifest as behavioral problems.

- Cross-environment consistency validation helps detect failures that passed in testing or staging environments by detecting divergence between environments early.
- Change events are captured with full attribution, making root-cause analysis faster when incidents do occur.

Level of risk exposed if this best practice is not established: High

Implementation guidance

First, determine your source of truth for configuration. If approved baselines live in a wiki, a shell history, or in the AWS console, then drift detection has nothing to compare against. Storing baselines as infrastructure as code (IaC) in [AWS CloudFormation](#) or the [AWS Cloud Development Kit \(AWS CDK\)](#) gives every deployment a reproducible reference point and makes the IaC definition the single artifact that authoritatively determines what resources should look like.

[AWS CloudFormation drift detection](#) reveals when deployed resources have diverged from their stack definitions. [AWS Config](#) rules add the runtime layer, monitoring agent infrastructure continuously and triggering automated remediation when deviations appear. [AWS CloudTrail](#) captures every configuration change event with full attribution, so when drift is detected, the team can determine exactly how a change was made without reconstructing events.

Behavioral configurations, system prompts, guardrail settings, tool permissions, and decision boundaries need a parallel track because they don't consistently sit in CloudFormation-manageable resources. A versioned configuration store with strict access controls and change notifications handles this layer. Production changes should require documented justification and sign-off.

The goal isn't to slow teams down but to send a prompt adjustment that alters downstream behavior through the same review as a code change. Teams using steering files in Kiro or equivalent can codify configuration standards so drift is less likely to be introduced at the source.

Scheduled cross-environment validation catches the slow category of drift that single-event detection misses. Snapshot the configuration of each environment on a cadence, compare the snapshots, and alert on any discrepancy that isn't explained by an approved change. This check reveals drift that accumulated gradually over months rather than arriving in a single event.

Implementation steps

1. **Define configuration baselines as IaC:** Store agent infrastructure definitions in [AWS CloudFormation](#) or [AWS CDK](#) under version control, with the IaC definition as the single source of truth.
2. **Configure drift detection:** Use [AWS CloudFormation drift detection](#) for infrastructure and [AWS Config](#) rules for agent-specific configurations (guardrail settings, model parameters) against approved baselines.
3. **Enable change event capture with full attribution:** Turn on [AWS CloudTrail](#) and route change events to alerting and automated remediation workflows.
4. **Version behavioral configurations:** Store prompts, guardrail settings, and decision boundaries in a versioned configuration store with access controls and mandatory approval workflows for production changes.
5. **Validate cross-environment consistency on a schedule:** Compare configuration snapshots across development, staging, and production, and alert on unexplained discrepancies.

Resources

Related best practices:

- [AGENTOPS02-BP01 Evolve agent prompts, tool calls, and configurations to reflect evolving business needs](#)
- [AGENTOPS02-BP03 Implement agent behavior versioning and rollback capabilities](#)
- [AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance](#)
- [AGENTREL08-BP01 Establish consistent configuration management practices](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Guidance for Agentic AI Operational Foundations on AWS](#)
- [Kiro Steering](#)
- [Evolving software delivery for agentic AI](#)

Related videos:

- [AWS re:Invent 2024 - Architecting scalable and secure agentic AI with AgentCore \(AIM431\)](#)

Related examples:

- [GitHub: Sample Agentic Platform on AWS](#)

Related services:

- [AWS Config](#)
- [AWS CloudTrail](#)
- [AWS CloudFormation](#)
- [Amazon CloudWatch](#)

AGENTOPS02-BP03 Implement agent behavior versioning and rollback capabilities

Teams with versioned behavior and tested rollback can recover in minutes when an agent behaves unexpectedly, while teams without spend hours debugging under pressure. Rapid reversibility improves your organization's ability to confidently iterate on an agentic system.

Desired outcome:

- Every agent behavioral configuration (system prompts, reasoning instructions, tool permissions, and decision boundaries) is versioned with a complete change history.
- You can roll back to any previous behavioral version within minutes when a change produces undesired outcomes.
- Rollback procedures are automated and tested regularly, not improvised during incidents.
- Staged rollouts limit the scope of impact of behavioral changes, and A/B testing supports data-driven comparison of variants before full deployment.

Common anti-patterns:

- Deploying behavioral changes to 100% of traffic immediately without staged rollout, maximizing the scope of impact when a change produces undesired outcomes.

- Operating without a defined behavioral baseline, the last known-good configuration, so rollback becomes a manual search for which previous version was stable.
- Treating prompt changes as low-risk because they don't involve code changes, skipping evaluation and staged rollout for modifications that can fundamentally alter agent behavior.
- Running A/B tests without statistical discipline, making deployment decisions from noise rather than signal.

Benefits of establishing this best practice:

- Systematic versioning and rollback create a safety net that lets teams iterate on agent behavior confidently, knowing any change can be reversed quickly.
- A/B testing frameworks and behavioral baselines provide the empirical foundation for continuous improvement, validating that each iteration produces measurable gains.
- Staged rollouts limit the users affected by a regression, giving the team detection and correction time before full exposure.
- Change impact assessment ties quality metric shifts to specific behavioral versions, making attribution direct instead of inferred.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock Prompt Management](#) handles prompt-based configurations with built-in semantic versioning, metadata, and integration with evaluation. Non-prompt configurations (tool permissions, decision boundaries, and escalation thresholds) should be stored in a versioned configuration store with change tracking. Each version should carry a semantic version number, a change description, an author, and a reference to its evaluation results.

A behavioral baseline is a version that the team has explicitly designated as known-good, not just the previous version, because the previous version might have been shipped an hour ago and never proven stable. Rollback should restore the baseline, not the last change, unless the team has explicitly promoted that change to baseline status. Without a designated baseline, rollback requires searching through multiple versions to find a stable configuration.

Rollback itself should be automated and rehearsed. Design rollback as an automated workflow triggered by either manual approval or by automated quality threshold violations from [Amazon](#)

[CloudWatch](#) alarms. The target time-to-restore should be under five minutes for behavioral changes. Anything longer means the workflow has too many manual steps or too many dependencies that aren't pre-staged. Exercise the rollback quarterly so the procedure stays current with the runtime. For example, a rollback that was written six months ago and never run is a rollback that may not work.

Staged rollout limits the scope of impact before rollback is ever needed. [Amazon Bedrock AgentCore Runtime](#) endpoint-based weighted routing makes this straightforward. Start a new behavioral version at 5–10% of traffic, monitor quality metrics, and promote only when the signal is clean. A/B testing uses the same machinery for different ends, splitting traffic between variants to measure which performs better. The critical additions are per-variant metrics in CloudWatch and statistical significance testing before deployment decisions. Document the evaluation criteria and results alongside the behavioral version record so the comparison is reproducible.

Perform impact assessments where you correlate version deployment timestamps with changes in quality metrics to attribute metric shifts to specific behavioral updates. When a metric moves, the team should quickly determine which version caused the issue as opposed to pattern-matching different dashboards to find the problematic version.

Implementation steps

1. **Enable semantic versioning for prompts:** Configure [Amazon Bedrock Prompt Management](#) with metadata fields for change description, author, and evaluation results on every version.
2. **Version non-prompt configurations:** Store tool permissions, decision boundaries, and escalation thresholds in a versioned configuration store with change tracking and notifications.
3. **Designate behavioral baselines:** Tag and document the last known-good configuration for each agent as the rollback target, distinct from the most recent version.
4. **Automate rollback:** Build workflows that restore baselines within five minutes, triggered by manual approval or automated quality threshold violations from [Amazon CloudWatch](#) alarms.
5. **Configure staged rollout:** Use [Amazon Bedrock AgentCore Runtime](#) endpoint-based weighted routing starting at 5–10% traffic for new behavioral versions, with automated promotion gates.
6. **Set up A/B testing infrastructure:** Capture per-variant metrics in [Amazon CloudWatch](#) with statistical significance tracking before deployment decisions.
7. **Correlate deployments to metric shifts:** Record deployment timestamps alongside quality metric trends so changes can be attributed to specific versions.

Resources

Related best practices:

- [AGENTOPS02-BP01 Evolve agent prompts, tool calls, and configurations to reflect evolving business needs](#)
- [AGENTOPS02-BP02 Implement configuration drift detection and remediation](#)
- [AGENTOPS03-BP02 Implement CI/CD pipelines tailored to agentic system deployment \(AgentOps\)](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Evolving software delivery for agentic AI](#)
- [Deploy AI agents on Amazon Bedrock AgentCore using GitHub Actions](#)

Related videos:

- [AWS re:Invent 2024 - Architecting scalable and secure agentic AI with AgentCore \(AIM431\)](#)
- [AWS re:Invent 2024 - Amazon Bedrock Agents and AgentCore Design Patterns \(TNC322\)](#)

Related examples:

- [GitHub: Sample Agentic Platform on AWS](#)
- [GitHub: Amazon Bedrock Samples, GenAI Quick-Start PoCs](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTOPS02-BP04 Maintain feedback control loops for continuous improvement

Agents that improve in step with real-world usage outperform agents frozen at deployment. A working feedback loop connects quality signals, user feedback, behavioral cues, and business outcomes to prioritized improvement actions.

Desired outcome:

- You collect and correlate agent performance data, user feedback, and business outcome metrics systematically, not through ad-hoc surveys.
- Feedback loops operate continually, detecting quality trends in near real time rather than through quarterly reviews.
- Improvement actions are tracked from identification through implementation and validation.
- Feedback signals are attributable to specific agent versions, so teams know which improvements are responding to which problems.

Common anti-patterns:

- Collecting user feedback (like thumbs up and down or ratings) without connecting it to specific agent behaviors or prompt versions, making it impossible to attribute quality changes to improvements.
- Relying solely on periodic manual reviews rather than continuous automated feedback processing, allowing quality degradation to persist for weeks before detection.
- Collecting feedback data without a defined process for turning insights into improvement actions, creating a growing backlog of signals that never translate into agent changes.
- Mixing signal types into a single bucket, so a surge in automated quality alerts drowns out a handful of high-severity user reports that deserve immediate attention.

Benefits of establishing this best practice:

- Structured feedback turns operational data into a continuous source of improvement signals, so agents evolve in response to real usage rather than staying static after deployment.
- Feedback-driven prioritization directs development effort toward changes with the greatest measurable impact.

- Trend tracking over time reveals patterns (data drift, concept drift, and scope drift) that inform targeted refinement rather than scattershot tweaking.
- Improvement validation gives the team evidence that each change delivered the expected gain.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Feedback loops should watch more than one signal to be truly useful.

Automated quality metrics from [Amazon Bedrock Evaluations](#) and [Amazon CloudWatch](#) show measurable shifts in output quality.

To watch subjective perception, consider checking:

- Explicit user feedback
- Thumbs up and down
- Ratings
- Free-text comments

To determine whether users are finding what they need, consider checking:

- Implicit behavioral signals
- Task abandonment
- Escalation rates
- Retry patterns

To determine if the agent is adhering to your organization's goals, consider checking:

- Business outcome metrics
- Conversion rate
- Resolution time
- Customer satisfaction

Each channel catches failures the others miss, so collecting all four of these metric pathways and routing them through a unified processing pipeline is the minimum viable design.

Use event-driven ingestion to keep your pipeline scalable. [Amazon EventBridge](#) or an equivalent event bus takes feedback events from every channel and routes them to a processing layer that classifies by type (quality issue, capability gap, tool failure, behavioral misalignment), severity, and affected component. Storing processed feedback in [Amazon DynamoDB](#) with indexing by agent, feedback type, and time period makes trend analysis and querying practical instead of painful.

Consider implementing severity-based routing to avoid drowning your teams in constant alerts. High-severity feedback, a user reporting the agent did something dangerous, or a sudden drop in a quality metric goes straight to an immediate-review queue. Lower-severity feedback aggregates into batch reviews that surface patterns over days rather than requiring immediate reactions.

Verify that you have an effective improvement tracking workflow. To keep your feedback process useful and actionable, you need:

- A durable workflow
- Identification
- Root cause analysis
- Improvement design
- Implementation
- Validation
- Correlation to the specific feedback that prompted the action
- Metrics compared before and after each change

Validation is the step most often skipped, and the one that tells the team whether an improvement was truly effective.

Dashboards help you address visibility of both feedback and improvements. Feedback trends alongside improvement outcomes provide a clear view of whether the agent's quality trajectory is rising, flat, or falling, and which improvements are responsible for each inflection.

Implementation steps

1. **Implement multi-channel feedback collection:** Cover automated quality metrics (through [Amazon Bedrock Evaluations](#) and [Amazon CloudWatch](#)), explicit user feedback, implicit behavioral signals, and business outcome metrics.
2. **Classify feedback at ingestion:** Categorize by type (quality issue, capability gap, tool failure, behavioral misalignment), severity, and affected component.

3. **Store processed feedback for trend analysis:** Use [Amazon DynamoDB](#) indexed by agent, feedback type, and time period.
4. **Route by severity:** Send high-severity feedback to immediate review queues through [Amazon EventBridge](#). Aggregate lower-severity items for batch review.
5. **Track improvements end to end:** Build a workflow that moves each item from identification through root-cause analysis, implementation, and validation, with metrics compared before and after.
6. **Build visibility into trends and outcomes:** Create dashboards that show feedback trends, improvement outcomes, and quality trajectory over time.

Resources

Related best practices:

- [AGENTOPS02-BP01 Evolve agent prompts, tool calls, and configurations to reflect evolving business needs](#)
- [AGENTOPS05-BP02 Monitor agent behavior patterns and detect anomalies](#)
- [AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Evaluating AI agents: Real-world lessons from building agentic systems at Amazon](#)
- [Guidance for Agentic AI Operational Foundations on AWS](#)

Related videos:

- [AWS re:Invent 2024 - Elevate application and generative AI observability \(COP326\)](#)
- [AWS re:Invent 2024 - Responsible generative AI: Evaluation best practices and tools \(AIM342\)](#)

Related examples:

- [GitHub: Open Source Bedrock Agent Evaluation](#)
- [GitHub: Sample Bedrock Evaluation Adapter](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon CloudWatch](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon EventBridge](#)
- [Amazon DynamoDB](#)

Agent lifecycle and deployment processes

Agent lifecycle management extends beyond traditional software deployment to include behavioral evolution, capability updates, and operational governance. Agents require ongoing management as they adapt to changing business requirements, integrate new tools, and scale across different environments.

AGENTOPS03: How do you manage agent lifecycle and deployment processes?

Capability intent

- Every agent has a documented owner, lifecycle state, and set of transition criteria, from initial development through to clean decommissioning.
- Agent changes reach production through automated deployment pipelines with agent-specific quality gates, staged rollout, and automatic rollback on regression.
- Agent capacity adjusts to workload demand within per-environment boundaries, and deployments are kept right-sized through regular capacity reviews.
- The agent portfolio is visible in a single catalog with cross-team dependencies tracked, so teams discover existing capabilities before building new agents.
- Portfolio-level rationalization (retirement, consolidation, continued investment) is driven by evidence of utilization, cost, and business value rather than team-level preference.

Maturity levels

These levels summarize what each stage of maturity looks like for agent lifecycle and deployment processes as a whole.

Level	Name	What it looks like
1	Initial	Agent lifecycle is informal. Teams spin up agents on demand, often straight to production, with no documented stages, no registry, and no designated owners. Deployments are manual. There is no agent-specific CI/CD or behavioral evaluation. Scaling is static or per-workload and revisited only when something breaks. No portfolio-level visibility exists across teams.
2	Emerging	A lifecycle with defined stages exists on paper and most teams follow it. Agents are tracked in a team-level registry. Basic CI/CD runs unit tests, and deployments use infrastructure as code templates. Per-environment scaling boundaries are set manually. Teams can list the agents they own, but there is no cross-team catalog or dependency view.

Level	Name	What it looks like
3	Defined	Lifecycle stages are standardized across the organization with validation gates at each transition. AgentOps pipelines include agent-specific evaluation through Amazon Bedrock Evaluations and staged rollout through Amazon Bedrock AgentCore Runtime versioning. Auto-scaling is driven by agent-specific metrics rather than raw CPU or memory. A shared agent catalog is emerging, and the first quarterly portfolio reviews are held.

Level	Name	What it looks like
4	Proactive	Lifecycle transitions are fully automated, including emergency termination switch and decommissioning runbooks. CI/CD behavioral gates block release when task accuracy or hallucination thresholds regress, with automated rollback from Amazon CloudWatch alarms. Scaling policies, stored centrally, adjust as agents move between lifecycle stages. AWS Agent Registry with semantic capability search and Amazon EventBridge dependency notifications is the default path for new agent creation.

Level	Name	What it looks like
5	Optimized	Lifecycle, deployment, scaling, and portfolio management are policy-codified and continuously refined from operational data. A dependency graph backs impact analysis before any agent change. Portfolio rationalization (consolidation, retirement, reinvestment) runs as a regular operating rhythm with business-value measurement. Governance overhead scales sub-linearly with agent count, and the organization contributes its patterns back to the industry.

Common issues to watch for

- Agents accumulate in production without documented owners or lifecycle states, so retirement and incident response stall because nobody knows who is accountable for each agent.
- Agent deployments rely on traditional software CI/CD without agent-specific behavioral evaluation, letting prompt or tool regressions reach users that unit tests would never catch.
- Scaling is configured identically across environments, either wasting cost in development or degrading production latency during traffic spikes.
- Teams build redundant agents in parallel because the organization lacks a shared catalog and discovery path, inflating cost and operational burden without adding capability.
- Cross-team agent dependencies are undocumented, so deprecating or modifying an upstream agent causes cascading failures in downstream agents that were silently consuming it.

Best practices

- [AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance](#)
- [AGENTOPS03-BP02 Implement CI/CD pipelines tailored to agentic system deployment \(AgentOps\)](#)
- [AGENTOPS03-BP03 Implement agent-specific scaling policies and capacity planning](#)
- [AGENTOPS03-BP04 Implement organizational agent portfolio management and governance at scale](#)

AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance

An agent portfolio without lifecycle discipline becomes a graveyard of undocumented services with forgotten owners. Explicit lifecycle stages, named SME ownership, and clean decommissioning keep the portfolio tractable as it grows from a handful of agents to dozens or hundreds.

Desired outcome:

- Every agent has a documented lifecycle state (development, pilot, production, deprecated, and decommissioned) with defined transition criteria.
- Onboarding follows a standardized provisioning process that configures required resources, permissions, and monitoring before an agent handles production traffic.
- Decommissioning cleanly removes retired agents, no orphaned resources, dangling permissions, or undocumented dependencies left behind.
- Each agent has a named SME owner accountable for its behavior, performance, and eventual retirement.

Common anti-patterns:

- Deploying agents to production without a defined lifecycle state or designated owner, so no one is accountable when behavior needs attention.
- Operating without decommissioning procedures, leaving retired agents running with active permissions and consuming resources long after they were replaced.
- Skipping the pilot stage and pushing agents from development directly to full production, missing the chance to validate behavior under real traffic with enhanced monitoring.
- Treating the agent registry as a one-time artifact that nobody updates once the agent is live.

Benefits of establishing this best practice:

- Standardized lifecycle procedures produce consistent provisioning, operation, and retirement, reducing operational complexity as the portfolio grows.
- Documented lifecycle states and transition criteria create an auditable record of each agent's history for compliance and governance.
- Named owners accelerate incident response. When an agent misbehaves, the team knows who to engage without a search.
- Clean decommissioning helps prevent the slow accumulation of abandoned resources that becomes a cost and security problem over time.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Five stages cover the operational arc of almost any agent:

1. Development (under active development, not serving production traffic)
2. Pilot (limited production use with enhanced monitoring and cost validation)
3. Production (full deployment with standard operational procedures)
4. Deprecated (scheduled for decommissioning, no new integrations)
5. Decommissioned (removed from service, resources cleaned up)

Each transition should carry explicit criteria, required approvals, validation gates, and documentation requirements, so stage changes are decisions rather than drift.

Pilot validates economic viability and identifies issues before full deployment, reducing the cost of addressing problems. For teams using spec-driven development with tools like Kiro, the spec workflow produces the documentation needed for lifecycle governance as a byproduct. This is a useful side effect worth using rather than rebuilding.

An agent registry is a durable artifact that makes this process coherent. It should track agent ID, lifecycle state, owner, dependencies, capabilities, and operational metadata. Without a registry, lifecycle state exists only in people's heads, making it difficult to track and manage consistently across the organization. The registry becomes the input for portfolio reviews, decommissioning dependency analysis, and emergency response.

Emergency lifecycle transitions deserve their own processes. Automated emergency termination switch mechanisms allow immediate revocation of an agent's permissions and halting of operations, enabling rapid response to operational issues. The decommissioning runbook does similar work for the planned case. It removes resources, revokes permissions, updates the registry, and notifies dependent systems as automated steps rather than as checklist items. For agents built through no-code platforms like [Amazon Quick Suite](#), the same lifecycle rules apply. The registry tracks them, portfolio reviews consider them, and decommissioning cleans them up.

Implementation steps

1. **Document the five lifecycle stages:** Specify transition criteria, required approver roles, and validation gates for each stage.
2. **Build the agent registry:** Track agent ID, lifecycle state, owner, dependencies, capabilities, and operational metadata in a durable store.
3. **Automate lifecycle state transitions:** Validate criteria, trigger stage-specific actions, and record transitions with attribution, deployments, permission changes, monitoring setup, and decommissioning steps.
4. **Create standardized provisioning templates:** Configure required resources, permissions, and monitoring automatically so new agents enter production with a consistent baseline.
5. **Implement emergency termination switch and decommissioning runbooks:** Include dependency analysis before running so decommissioning doesn't break upstream consumers.
6. **Establish quarterly portfolio reviews:** Identify agents for deprecation or decommissioning, including those built with no-code platforms like [Amazon Quick](#).

Resources

Related best practices:

- [AGENTOPS01-BP01 Establish well-defined agent roles, responsibilities, and success criteria](#)
- [AGENTOPS03-BP02 Implement CI/CD pipelines tailored to agentic system deployment \(AgentOps\)](#)
- [AGENTOPS03-BP03 Implement agent-specific scaling policies and capacity planning](#)
- [AGENTOPS02-BP02 Implement configuration drift detection and remediation](#)
- [AGENTCOST06-BP02 Cost optimize versioning and deployment through efficient artifact management](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Focus area 5: Manage the lifecycle](#)
- [Evolving software delivery for agentic AI](#)
- [Preparing the business for agentic AI at scale](#)
- [Kiro Specs](#)
- [Operationalizing Agentic AI Part 1: A Stakeholder's Guide](#)

Related videos:

- [AWS re:Invent 2024 - Agents in the enterprise: Best practices with AgentCore \(AIM3310\)](#)
- [AWS re:Invent 2024 - Cox Automotive's Blueprint for Agentic AI on AgentCore \(IND3329\)](#)
- [AWS re:Invent 2024 - Bridging from POC to production: Intro to AgentCore \(AIM2204\)](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTOPS03-BP02 Implement CI/CD pipelines tailored to agentic system deployment (AgentOps)

Manual agent deployments and informal testing can keep a project stuck in the pilot phase. An agent-aware pipeline, with behavioral evaluation gates, staged rollout, and automated rollback can help your organization realize the goal of daily deployment of behavioral improvements.

Desired outcome:

- Agent deployments run fully through CI/CD with agent-specific validation gates for prompt quality, tool integration correctness, behavioral regression, and security.

- Deployment strategies (blue/green, canary) limit the scope of impact when a regression does slip through.
- Automated rollback restores the previous version within minutes if quality thresholds are exceeded.
- Infrastructure is defined as code so deployments are reproducible and environments stay consistent.

Common anti-patterns:

- Deploying agent changes through manual console clicks or one-off scripts without automated validation gates, making deployments inconsistent and error-prone.
- Running only traditional unit tests without agent-specific behavioral evaluation (prompt quality, tool selection accuracy, hallucination rate), missing regressions that unit tests can't detect.
- Deploying directly to production without staged rollout (canary, blue/green), maximizing the scope of impact of any regression.
- Treating rollback as a theoretical capability that has never been exercised, so the first time anyone uses it is during an incident.

Benefits of establishing this best practice:

- Automated pipelines help every deployment follow the same validated path regardless of who starts it, reducing deployment inconsistency.
- Behavioral validation gates provide empirical evidence that each deployment meets quality standards before reaching production.
- Staged rollout and automated rollback compress incident response time from hours to minutes when regressions appear.
- Infrastructure as code makes deployments reproducible across environments, removing a common source of failures.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Agent CI/CD shares most of its structure with software CI/CD, with one substantive addition: behavioral evaluation. The stages that fit most agent workloads are:

- Source (code, prompts, configurations, and evaluation datasets)
- Build (package artifacts and run unit tests)
- Evaluate (run behavioral evaluation through [Amazon Bedrock Evaluations](#))
- Security scan (prompt injection vulnerabilities and IAM scope)
- Deploy to production

Task completion accuracy, hallucination rate, and tool selection accuracy need explicit thresholds that block promotion when exceeded. Thresholds that are set too loose produce false passes, but thresholds that are set too tight block legitimate iteration. To calibrate, start with thresholds tuned to the current baseline, then tighten them as the agent's quality track record grows.

Production deployment uses [Amazon Bedrock AgentCore Runtime](#) for managed scaling, versioning, and observability. agentcore deploy pushes new versions, and endpoint-based weighted routing handles blue/green and canary patterns. [Amazon CloudWatch](#) alarms watch quality metrics post-deployment and trigger automated rollback when thresholds are exceeded. The same alarms that run during staged rollout double as rollback triggers. Infrastructure as code through [AWS CDK](#) or [AWS CloudFormation](#) helps make every resource reproducible.

A rollback procedure that has never been exercised is a procedure that may not work when the team needs it. Deliberate rollback drills during pipeline validation confirm the revert works before the team is depending on it.

Implementation steps

1. **Build the pipeline stages:** Configure source, build, behavioral evaluation, security scan, and production deployment stages with the appropriate tools for each.
2. **Set behavioral evaluation as a gate:** Integrate [Amazon Bedrock Evaluations](#) with task completion accuracy and hallucination rate thresholds that block promotion when exceeded.
3. **Deploy to [Amazon Bedrock AgentCore Runtime](#):** Use built-in versioning and endpoint-based weighted routing for blue/green or canary rollouts.
4. **Automate rollback on quality threshold exceedance:** Wire [Amazon CloudWatch](#) alarms to revert-deployment workflows so quality threshold violations trigger immediate revert.
5. **Version all deployment artifacts:** Tag each artifact set with the pipeline run ID for traceability, and store in a durable versioned store.
6. **Validate the full pipeline:** Deliberately trigger a rollback during pipeline validation to confirm revert procedures work before they are needed for real.

Resources

Related best practices:

- [AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance](#)
- [AGENTOPS02-BP03 Implement agent behavior versioning and rollback capabilities](#)
- [AGENTOPS06-BP03 Establish SME-driven validation and business approval workflows](#)
- [AGENTCOST06-BP02 Cost optimize versioning and deployment through efficient artifact management](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Evolving software delivery for agentic AI](#)
- [Deploy AI agents on Amazon Bedrock AgentCore using GitHub Actions](#)
- [Strands Agents](#)
- [CI/CD and automation for serverless AI](#)
- [Kiro Hooks](#)

Related videos:

- [AWS 2025 - Deploy Production-Ready Agents in 22 Minutes with AgentCore Runtime](#)
- [AWS 2025 - Deploy ANY AI Agent to Production in Minutes - AgentCore Tutorial](#)
- [AWS 2025 - Strands Agents Observability, Evaluation, & Deployment](#)
- [AWS re:Invent 2024 - Building AI Agents with Serverless, Strands, and MCP \(NTA405\)](#)
- [AWS re:Invent 2024 - Develop AI Agents faster with SageMaker AI Studio & AgentCore \(AIM388\)](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore, Lab 4: Deploy to Production](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)

- [AWS Cloud Development Kit \(AWS CDK\)](#)
- [Amazon CloudWatch](#)

AGENTOPS03-BP03 Implement agent-specific scaling policies and capacity planning

Scaling policies designed for typical web workloads don't fit agents. Model inference latency, tool dependency availability, and downstream service capacity all shape the right response to load, and a policy that ignores them either over-provisions during quiet hours or under-provisions during peaks.

Desired outcome:

- Agent compute scales dynamically in response to demand while respecting cost, performance, and governance constraints.
- Scaling decisions account for agent-specific factors: model inference latency, tool availability, and downstream service capacity.
- Per-environment scaling boundaries help prevent runaway scaling in development while preserving capacity headroom in production.
- Monthly capacity reviews keep deployments right-sized as usage patterns evolve.

Common anti-patterns:

- Using identical scaling configurations across all environments, either over-provisioning development or under-provisioning production during traffic spikes.
- Scaling based solely on CPU and memory utilization without considering agent-specific metrics like request queue depth or inference latency, missing the real bottleneck.
- Setting scaling policies once at deployment and never revisiting them as usage patterns evolve.
- Treating capacity planning as a quarterly finance exercise rather than an ongoing operational one, so policies fall out of step with reality.

Benefits of establishing this best practice:

- Scaling policies adapt to deployment context and agent compute model, keeping capacity appropriate as workloads move from prototype to production scale.

- Per-environment boundaries and centralized configuration make scaling behavior consistent, auditable, and governed across environments.
- Monthly reviews catch drift between configured capacity and actual usage patterns before it becomes a cost or latency problem.
- Agent-specific scaling metrics expose the real bottleneck, often downstream service capacity or model throttling, that generic metrics hide.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

[Amazon Bedrock AgentCore Runtime](#) handles the scaling question for most teams through its built-in consumption-based scaling and pricing model. AgentCore Runtime allocates compute automatically based on demand, so custom scaling policies are not required. For agents deployed outside AgentCore Runtime, on [AWS Lambda](#) or [Amazon ECS](#) for example, scaling triggers must be configured against agent-specific signals: request queue depth, average response latency, and concurrent invocation count. CPU and memory alone miss the mark because agents often wait on model inference or downstream tools rather than saturating local compute.

Configure environment-appropriate maximums to control cost in development and maintain performance in production. Development can run with permissive minimums and tight maximums. Cost is the first consideration, so the policy should expect spikes and throttle them. In production, maximums should be generous enough to absorb traffic bursts without latency degradation, and minimums should hold enough warm capacity to avoid cold starts during demand ramps. Staging is the midpoint between development and production, so maximums and minimums should match that as well.

Store scaling configurations centrally (like in a parameter store integrated with the agent registry) so boundaries adjust automatically when an agent transitions between lifecycle stages. A configuration store also gives the team a single place to audit and adjust policies, instead of chasing them through individual service consoles.

The operations team should perform monthly reviews, analyzing scaling event history, peak utilization patterns, and capacity headroom across the portfolio using [Amazon CloudWatch](#) metrics. The outputs are concrete:

- Agents to scale down
- Agents to increase ceilings for

- Demand forecasts for the upcoming period

Without this review, scaling policies drift from the workload they were tuned against. For fleet-level operational visibility including dashboards, anomaly detection, and behavioral monitoring, see [AGENTOPS05-BP05 Create workflow-specific dashboards for operational health](#).

Implementation steps

1. **Choose the right scaling foundation:** Deploy agents on [Amazon Bedrock AgentCore Runtime](#) for built-in scaling, or configure auto scaling policies with agent-specific metrics for non-Runtime deployments.
2. **Set per-environment boundaries:** Use permissive policies for development, moderate for staging, and production with higher minimums to absorb traffic spikes.
3. **Centralize scaling configurations:** Store policies in a parameter store integrated with the agent registry so boundaries adjust as agents transition between lifecycle stages.
4. **Review capacity monthly:** Analyze scaling events, peak utilization, and capacity headroom in [Amazon CloudWatch](#) to right-size deployments and forecast demand.

Resources

Related best practices:

- [AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance](#)
- [AGENTOPS03-BP02 Implement CI/CD pipelines tailored to agentic system deployment \(AgentOps\)](#)
- [AGENTOPS02-BP02 Implement configuration drift detection and remediation](#)
- [AGENTOPS05-BP04 Define and track KPIs for agent workflows](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Preparing the business for agentic AI at scale](#)
- [Introducing Amazon Bedrock AgentCore: Securely deploy and operate AI agents at any scale](#)
- [Securely launch and scale your agents and tools on Amazon Bedrock AgentCore Runtime](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Runtime](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTOPS03-BP04 Implement organizational agent portfolio management and governance at scale

A handful of agents built by one team is a project. Dozens of agents built by multiple teams is a portfolio, and portfolios need different operational mechanisms. Without cross-organizational visibility, teams build redundant agents, break each other's integrations, and lose track of which agents still earn their keep.

Desired outcome:

- A centralized catalog gives the organization a current view of every agent, owner, capabilities, dependencies, lifecycle state, cost profile, and business value.
- Teams discover existing agents before building new ones, so effort goes to capability gaps rather than duplicates.
- Cross-team dependencies are tracked and managed through coordinated deprecation processes.
- Quarterly portfolio reviews reveal underutilized agents for consolidation or retirement, keeping the portfolio aligned with business priorities.

Common anti-patterns:

- Allowing teams to build agents independently without checking for existing capabilities, creating redundant agents that duplicate development cost, infrastructure cost, and operational burden.
- Maintaining agent registries only within individual teams, so no one has the cross-organizational view needed to identify redundancy or assess overall system health.
- Deprecating or modifying agents without notifying dependent teams, causing cascading failures when orchestrators or delegating agents invoke agents that have changed.

- Treating agent creation as a no-justification-required activity, so the portfolio grows faster than the organization's ability to operate, monitor, and maintain it.
- Failing to measure business value relative to operational cost, reducing the risk of data-driven decisions about which agents warrant continued investment.

Benefits of establishing this best practice:

- Portfolio governance scales operational practices from individual agents to enterprise environments, so governance overhead grows sub-linearly with agent count.
- A centralized catalog with ownership, dependency tracking, and lifecycle state provides the auditable record needed for compliance and organizational accountability.
- Capability search before build reduces redundant development, freeing engineering effort for capability gaps.
- Quarterly reviews help prevent sprawl by revealing candidates for consolidation and retirement before the portfolio becomes unmanageable.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Portfolio governance extends the team-level registry into an organizational catalog that spans team boundaries (for more detail, see [AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance](#)). The fields added at this layer are the ones that support cross-team decisions:

- Owning team
- Business domain
- Upstream dependencies (agents or systems that invoke this agent)
- Downstream dependencies (agents, tools, or services this agent invokes)
- Cost-per-month (derived from CloudWatch and Cost Explorer)
- Business value indicators (task completion volume, business outcome metrics).

[AWS Agent Registry](#) provides the centralized catalog with built-in approval workflows, flexible metadata, and hybrid search (semantic and keyword) so cross-organizational queries run efficiently.

Enrich registry records with dependency metadata, cost attribution, and business value indicators so the catalog supports portfolio-level decisions beyond simple discovery.

The pre-creation review gate helps prevent duplicate builds. Teams searching for specific agents by purpose will not find one through keyword matching against agent names. [AWS Agent Registry](#) provides built-in hybrid search that combines semantic understanding with keyword matching, so natural-language capability queries reveal existing agents with overlapping capabilities. When overlap exists, the requesting team documents why the existing agent is insufficient before proceeding.

This applies to code-based agents (Strands Agents, LangGraph) and no-code agents built through [Amazon Quick Suite](#) alike. A lightweight CI/CD gate that checks the registry and flags potential duplicates is enough for most organizations, while heavy approval processes encourage bypass.

Implement cross-team dependency tracking as a managed practice. Agents declare their upstream and downstream dependencies at registration and update declarations when dependencies change. [Amazon EventBridge](#) publishes events when agents are deprecated, modified, or decommissioned so downstream teams receive advance notice. Agents exposed through [Amazon Bedrock AgentCore Gateway](#) benefit from Gateway's tool registration metadata, which automatically tracks which agents consume which tools and reduces manual declaration burden. A dependency graph in [Amazon Neptune](#) enables impact analysis for determining how a change to one agent will affect others.

Quarterly portfolio reviews help prevent gradual drift. The review assesses four dimensions:

1. Utilization (which agents are actively used and which are idle)
2. Cost efficiency (which agents deliver business value proportional to cost)
3. Redundancy (which agents overlap with others)
4. Health (which agents show elevated error rates, degraded performance, or stale configurations)

The catalog is the primary data source, enriched with [Amazon CloudWatch](#) metrics, [AWS Cost Explorer](#) attribution, and the dependency graph. Reviews produce concrete recommendations. These include agents to consolidate, deprecate, or invest in, and cross-team dependency risks to address.

Implementation steps

1. **Extend the agent registry into an organizational catalog:** Use [AWS Agent Registry](#) to catalog agents with metadata for owning team, business domain, upstream and downstream dependencies, cost-per-month, and business value indicators.
2. **Enable semantic capability search:** Use Agent Registry's built-in hybrid search so natural-language queries reveal overlapping capabilities before new development begins.
3. **Gate new agent creation on registry search:** Add a pre-creation CI/CD check that requires justification when overlapping agents exist.
4. **Track cross-team dependencies:** Use [Amazon EventBridge](#) to notify downstream teams when upstream agents are deprecated, modified, or decommissioned.
5. **Build a dependency graph:** Use [Amazon Neptune](#) to answer impact-analysis questions before agent modifications.
6. **Run quarterly portfolio reviews:** Assess utilization, cost efficiency, redundancy, and health, producing specific recommendations for consolidation, deprecation, and investment.

Resources

Related best practices:

- [AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance](#)
- [AGENTOPS03-BP02 Implement CI/CD pipelines tailored to agentic system deployment \(AgentOps\)](#)
- [AGENTOPS04-BP01 Implement tool registry and catalog management](#)
- [AGENTOPS05-BP04 Define and track KPIs for agent workflows](#)
- [AGENTCOST06-BP01 Implement lightweight discovery and registry for cost-effective collaboration](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Preparing the business for agentic AI at scale](#)
- [AI agents in enterprises: Best practices with Amazon Bedrock AgentCore](#)

Related videos:

- [AWS re:Invent 2024 - Agents in the enterprise: Best practices with AgentCore \(AIM3310\)](#)
- [AWS 2025 - AgentCore Registry: Discover, Govern, and Reuse AI Agents at Scale](#)
- [AWS re:Invent 2024 - Cox Automotive's Blueprint for Agentic AI on AgentCore \(IND3329\)](#)

Related services:

- [AWS Agent Registry](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon EventBridge](#)
- [Amazon Neptune](#)
- [Amazon CloudWatch](#)
- [AWS Cost Explorer](#)

Tool integration and management practices

Agents rely on external tools and services to complete work, and the way those integrations are managed determines whether agent behavior stays consistent as the tool inventory grows. Without a shared registry, standardized protocols, and common fallback behavior, every new tool adds operational burden and every tool failure risks broader agent outages.

AGENTOPS04: How do you establish tool integration and management practices?

Capability intent

- Agents discover available tools and their capabilities through a single authoritative catalog rather than hardcoded integrations or spoken knowledge.
- Tools are integrated through standardized protocols such as [Model Context Protocol \(MCP\)](#) and [Agent-to-Agent \(A2A\)](#), so agents interoperate across providers and across team boundaries.
- Tool invocations are authenticated, authorized to least privilege, and fully auditable at the service layer rather than per agent.
- Tool failures are contained and routed through documented fallback chains, so a single degraded dependency doesn't cascade into agent-wide outages.

- Tool inventory, version distribution, and per-tool reliability are monitored centrally, giving operators the visibility to evolve and retire tools without disrupting dependent agents.

Maturity levels

These levels summarize what each stage of maturity looks like for tool integration and management as a whole.

Level	Name	What it looks like
1	Initial	No centralized registry exists. Teams build agent-to-tool integrations point-to-point with each tool's interface. Authentication, error handling, and logging are per-tool and inconsistent. A single degraded tool often cascades into broader agent failures because no fallback plan exists, and tool usage is visible only through ad-hoc logs.
2	Emerging	A registry lists the tools in use with owner and version, even if enforcement is informal. Most new tools are exposed through a standard protocol, typically MCP through Amazon Bedrock AgentCore Gateway , rather than as bespoke integrations. Basic retry and timeout policies are applied per tool, and teams document the most common tool failures. Audit logging

Level	Name	What it looks like
		is collected but not currently centralized.
3	Defined	Tools are exposed through AgentCore Gateway or an equivalent registry and discovered by agents through semantic search. Protocol-based integration using MCP and A2A in AgentCore Runtime is the default for new tools. Per-tool authentication, authorization with Cedar policies in Policy in AgentCore , and audit logging are enforced centrally at the service layer. Fallback chains and circuit-breaker cutoffs exist for the most critical tools, and per-tool reliability metrics are tracked in a shared dashboard.

Level	Name	What it looks like
4	Proactive	The registry is the single source of truth for tool definitions, versions, ownership, and deprecation timelines, and registration is a gate on production use. Standardized protocols and least-privilege policies are enforced across the organization, with Cedar policy reviews part of the change process. Fallback chains, exponential backoff and jitter, and automatic cutoffs are applied consistently based on a shared error taxonomy. Tool telemetry is instrumented end-to-end through Amazon Bedrock AgentCore Observability, with alarms and runbooks owned by tool teams.

Level	Name	What it looks like
5	Optimized	The tool inventory is governed by policy-as-code. New tools must pass registry, protocol, policy, and observability gates before any agent can invoke them. Protocol and policy choices are continuously validated and tightened using closed-loop feedback from production telemetry. Fallback chains adjust automatically based on historical reliability, and the organization contributes improvements back to the open protocol specifications it depends on. Tool deprecation is routine and low-risk.

Common issues to watch for

- Teams build point-to-point tool integrations without a shared registry or protocol, producing duplicate tools and inconsistent documentation that agents can't reliably discover or use across team boundaries.
- Tools are registered without a documentation standard or a named owner, so agents encounter undocumented parameter schemas and error codes only at runtime.
- Tool authorization and audit logging are implemented per tool instead of centrally, leaving gaps in which agent invoked which tool and undermining compliance reviews.
- Retry policies are identical for every tool, so permanently failed or rate-limited dependencies are hammered with aggressive retries while transient issues receive too few.
- Tools are deprecated or versioned without notifying dependent agents, so runtime failures surface only when the old version disappears.

Best practices

- [AGENTOPS04-BP01 Implement tool registry and catalog management](#)
- [AGENTOPS04-BP02 Establish standardized tool integration protocols \(MCP, A2A\)](#)
- [AGENTOPS04-BP03 Develop fallback behavior and error handling for tool invocations](#)

AGENTOPS04-BP01 Implement tool registry and catalog management

Agents that can't discover tools end up with hardcoded integrations, inconsistent documentation, and a maintenance tax that grows with every new capability. A centralized registry makes the tool catalog queryable, versioned, and governable.

Desired outcome:

- A centralized tool registry provides a single authoritative source for tool capabilities, current versions, and operational status.
- Tool documentation is standardized and complete so agents can select and invoke tools without guesswork.
- Tool versioning and compatibility tracking help prevent agents from invoking deprecated or incompatible versions.
- Tool deprecation procedures remove sunset tools cleanly without disrupting dependent agents.

Common anti-patterns:

- Allowing each team to manage tool definitions independently without a shared registry, producing duplicate tools, inconsistent documentation, and agents that can't discover tools built by other teams.
- Registering tools without documentation standards, leaving agents to guess at parameter formats and error codes through trial and error.
- Deprecating tools without notifying dependent agents, causing runtime failures when agents attempt to invoke tools that no longer exist.
- Letting the registry drift out of step with reality, so health status in the registry contradicts actual tool availability.

Benefits of establishing this best practice:

- A single source of truth for tool capabilities and status means all agents discover and use tools consistently, and updates propagate reliably.
- Centralized visibility into the tool catalog lets teams track availability, version distribution, and dependency relationships across the agent portfolio.
- Documentation standards reduce invocation errors by giving agents complete, structured information about each tool.
- Deprecation workflows give dependent agents time to migrate before tools are removed, helping prevent outages.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Start with what [Amazon Bedrock AgentCore Gateway](#) provides before building custom. AgentCore Gateway includes semantic tool discovery and MCP server capabilities that handle cataloging, versioning, and discovery without custom infrastructure. For many teams this covers the registry requirement completely. Evaluate AgentCore Gateway against the discovery workflow you need, including capability search, version tracking, and dependency metadata, before investing in parallel infrastructure.

A gap for most organizations is documentation standards, not the registry mechanism. A tool entry can be invoked incorrectly without a purpose statement, parameter schemas, error codes, rate limits, and authentication requirements. Enforce completeness as a gate in the onboarding process rather than as a post-registration suggestion. The cost of rejecting an undocumented tool during registration is lower than the cost of debugging misuse after deployment.

Semantic versioning with compatibility guarantees helps agents and tools evolve at different speeds. A new version of a tool should not silently break agents that were written against the previous version, and a new agent should not be blocked on tools that have not exposed the version it needs. Maintain multiple active versions during transitions, and implement deprecation workflows that notify dependent agents before they are deprecated. Health checks update tool operational status so that the registry is updated with realistic data.

Implementation steps

1. **Evaluate [Amazon Bedrock AgentCore Gateway](#) as the primary registry:** Use its semantic tool discovery and MCP server capabilities, and supplement with custom infrastructure only where a specific capability is missing.

2. **Define tool documentation standards:** Require purpose, parameter schemas, error codes, rate limits, and authentication requirements. Enforce completeness as a gate in the tool onboarding process.
3. **Implement semantic versioning:** Use compatibility guarantees and maintain multiple active versions during transitions so agents and tools can evolve independently.
4. **Configure health checks and deprecation workflows:** Update operational status automatically and notify dependent agents before sunset dates.

Resources

Related best practices:

- [AGENTOPS04-BP02 Establish standardized tool integration protocols \(MCP, A2A\)](#)
- [AGENTOPS04-BP03 Develop fallback behavior and error handling for tool invocations](#)
- [AGENTOPS01-BP01 Establish well-defined agent roles, responsibilities, and success criteria](#)
- [AGENTSEC02-BP03 Maintain approved tool registry with security assessments](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Agentic AI frameworks, platforms, protocols, and tools on AWS](#)
- [AI agents in enterprises: Best practices with Amazon Bedrock AgentCore](#)
- [Introducing Amazon Bedrock AgentCore Gateway: Transforming enterprise AI agent tool development](#)
- [Guidance for Agentic AI Operational Foundations on AWS](#)

Related videos:

- [AWS re:Invent 2024 - Scale agent tools with AgentCore Gateway \(AIM3313\)](#)
- [AWS re:Invent 2024 - Build agentic workflows with third-party agents and tools \(AIM3311\)](#)
- [AWS re:Invent 2024 - Modernize containers for AI agents using AgentCore Gateway \(CNS422\)](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTOPS04-BP02 Establish standardized tool integration protocols (MCP, A2A)

Point-to-point integrations between every agent and every tool produce a maintenance burden that grows with the number of integrations. Standardized protocols like MCP and A2A replace this with a shared contract, which defaults to interoperability instead of tools that are custom-made for specific integrations.

Desired outcome:

- Agents integrate with tools through standardized protocols (MCP for tool invocation, A2A for agent-to-agent) that support interoperability, consistent behavior, and portability across providers.
- Tool invocations run with secure patterns: least-privilege access, consistent error handling, and complete audit logs.
- Agents invoke tools reliably across varying network conditions and tool availability, with fallback mechanisms that maintain service continuity.
- Error handling follows a standardized taxonomy so every agent responds to transient, permanent, and authorization failures the same way.

Common anti-patterns:

- Building custom point-to-point integrations for every agent-tool pair instead of adopting standard protocols, creating a maintenance burden that grows with scale.
- Implementing tool invocations without standardized error handling, so each agent handles failures differently and inconsistently.
- Skipping authentication and audit logging for tool invocations, making it impossible to trace which agent invoked which tool or whether it was authorized.
- Treating protocol versioning as an afterthought, so a tool upgrade silently breaks the agents that depended on the previous version.

Benefits of establishing this best practice:

- Standardized tool integration with least-privilege access enforces operational boundaries at the invocation layer, not just at the agent's internal logic.
- Audit logging of every tool invocation creates the evidentiary record required for compliance and security reviews.
- Shared error handling patterns mean operators debug tool failures the same way across every agent.
- Protocol-based integration lets tool providers change backends without breaking agent consumers, and the reverse.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock AgentCore Gateway](#) is the primary integration layer for MCP-compatible tool access. It provides managed authentication, authorization, and tool discovery through a standardized interface, so agents don't each reimplement those pieces. For agent-to-agent communication, [Amazon Bedrock AgentCore Runtime](#) supports A2A protocol endpoints with agent discovery through Agent Cards, task lifecycle management, and structured message exchange. The two together cover most integration surfaces without custom infrastructure.

Least privilege enforcement needs to happen at the protocol layer, not at the application layer.

[Amazon Bedrock AgentCore Policy](#) applies Cedar policies that scope each agent's tool permissions at the Gateway boundary. Agents can invoke only the tools their policy allows, regardless of what their internal code tries to do. The check runs at traffic time, not at review time. Establish audit logging through [AgentCore Gateway](#), which produces compliance records without requiring custom instrumentation.

Error handling benefits a taxonomy of transient errors (retry with exponential backoff and jitter), permanent errors (fail gracefully), and authorization errors (escalate to human review). Each class calls for different agent behavior, and conflating them, such as retrying a permanent error or escalating a transient timeout, produces the wrong response at scale. For critical tools, implement fallback chains that attempt alternatives when the primary tool is unavailable. Monitor per-tool error rates and latency through [Amazon Bedrock AgentCore Observability](#), and configure alarms so degradation is detected before it becomes an incident.

Protocol versioning through capability negotiation preserves backward compatibility as protocols evolve. Version mismatches should result in the older side operating at its known capability rather than failing, and both sides should declare supported versions during handshake.

Implementation steps

1. **Expose tools through [Amazon Bedrock AgentCore Gateway](#):** Publish MCP server capabilities with [Amazon Bedrock AgentCore Policy](#) enforcing per-agent access controls.
2. **Implement A2A protocol endpoints:** Use [Amazon Bedrock AgentCore Runtime](#) for standardized inter-agent communication.
3. **Define protocol versioning strategies:** Use capability negotiation so older and newer sides interoperate at the common supported version.
4. **Implement standardized error handling:** Apply the transient/permanent/authorization taxonomy and fallback chains for critical tools across every agent.
5. **Monitor per-tool health through [Amazon Bedrock AgentCore Observability](#):** Track error rates and latency, and configure alarms for proactive detection.

Resources

Related best practices:

- [AGENTOPS04-BP01 Implement tool registry and catalog management](#)
- [AGENTOPS04-BP03 Develop fallback behavior and error handling for tool invocations](#)
- [AGENTOPS01-BP01 Establish well-defined agent roles, responsibilities, and success criteria](#)
- [AGENTPERF06-BP02 Implement efficient tool invocation patterns](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Open Protocols for Agent Interoperability Part 1: Inter-Agent Communication on MCP](#)
- [Open Protocols for Agent Interoperability Part 4: Inter-Agent Communication on A2A](#)
- [Introducing agent-to-agent protocol support in Amazon Bedrock AgentCore Runtime](#)
- [Open Protocols for Agent Interoperability Part 2: Authentication on MCP](#)
- [Agentic AI frameworks: Protocol-based tools](#)

Related videos:

- [AWS re:Invent 2024 - Building Scalable, Self-Orchestrating AI Workflows with A2A and MCP \(DEV415\)](#)

- [AWS 2025 - AgentCore Deep Dive: Gateway](#)
- [AWS 2025 - Integrating MCP Tools with Strands Agents](#)

Related workshops:

- [Diving Deep into Bedrock AgentCore, Gateway](#)

Related tools:

- [Strands Agents](#)

Related services:

- [AWS Identity and Access Management](#)
- [Amazon Bedrock AgentCore](#)

AGENTOPS04-BP03 Develop fallback behavior and error handling for tool invocations

Tools failing is a reality in modern architectures. The question is whether the agent degrades gracefully to an alternative, falls back to a manual process, or returns an error and halts. Well-designed fallback chains and automatic cutoffs keep tool failures from becoming agent failures.

Desired outcome:

- When tools fail or become unavailable, agents degrade gracefully to alternative tools, degraded-mode operations, or manual process escalation.
- Automatic cutoffs help prevent cascading failures from propagating through the agent environment.
- Retry strategies with exponential backoff handle transient errors transparently without amplifying load on degraded systems.
- Per-tool success rates, latency, and error patterns are monitored in real time so degradation is detected before it impacts users.

Common anti-patterns:

- Implementing identical retry strategies for all tools regardless of failure characteristics, applying aggressive retries to permanently failed tools or insufficient retries to transiently degraded ones.
- Operating without automatic cutoffs, so agents repeatedly invoke degraded tools that consistently time out, wasting time and resources on calls unlikely to succeed.
- Having no fallback path when a tool fails, forcing the agent to return an error to the user or halt the workflow entirely.
- Treating tool runbooks as optional documentation, so operators start from scratch on each incident.

Benefits of establishing this best practice:

- Per-tool monitoring with error pattern analysis gives deep visibility into tool reliability across the catalog.
- Operational runbooks help drive consistent incident response, reducing mean time to resolution and helping prevent ad-hoc responses that introduce new problems.
- Automatic cutoffs break cascading failures at their source rather than allowing them to propagate across agents.
- Graceful degradation through fallback chains maintains business continuity during partial outages.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Automatic cutoffs are the single most effective control for helping prevent cascading failure. When a tool starts returning errors above a threshold, continuing to invoke it compounds the problem. The agent wastes time, the tool gets more load, and upstream latency climbs while nothing is accomplished. An automatic cutoff transitions the tool from normal operation to blocked when error rates exceed a threshold (for example, 50% errors in a 60-second window) or when timeouts exceed a threshold (for example, 5 consecutive timeouts). It transitions to probing after a configurable recovery interval (for example, 30 seconds). Apply the state machine to every external dependency rather than the ones that have already caused incidents.

Retry and timeout policies should be per-tool rather than global. Exponential backoff with jitter handles transient errors without amplifying the problem. Fallback chains, like primary to secondary to degraded-mode to manual escalation, give the agent a path forward when the primary tool is

unavailable for critical operations. The fallback order should live in the tool registry so it evolves with the tool inventory rather than embedded in each agent's code.

[Amazon Bedrock AgentCore Observability](#) tracks invocation count, success rate, error rate by type, and latency percentiles on a per-tool basis. Composite alarms correlate multiple metrics. A tool with rising latency and increasing timeout rate is a tool approaching cutoff, regardless of whether either signal would fire an alarm individually. The composite view often catches degradation earlier than single-metric thresholds would.

Develop operational runbooks for the top five most common tool failure scenarios, and generate weekly SLA reports shared with tool owners so reliability trends stay visible.

Implementation steps

1. **Implement automatic cutoffs:** Configure error rate and timeout thresholds for the transition from normal to blocked, plus probing recovery timeouts, for every external dependency.
2. **Define per-tool timeout and retry policies:** Use exponential backoff with jitter appropriate to each tool's failure characteristics.
3. **Build fallback chains for critical tools:** Store fallback order in the tool registry so chains evolve with the tool inventory rather than being hardcoded in each agent.
4. **Monitor per-tool health through [Amazon Bedrock AgentCore Observability](#):** Use composite alarms that correlate multiple metrics for earlier detection.
5. **Develop operational runbooks for common tool failure scenarios:** Include diagnostic steps and escalation paths so operators start from a known baseline.

Resources

Related best practices:

- [AGENTOPS04-BP01 Implement tool registry and catalog management](#)
- [AGENTOPS04-BP02 Establish standardized tool integration protocols \(MCP, A2A\)](#)
- [AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations](#)
- [AGENTPERF06-BP02 Implement efficient tool invocation patterns](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)

- [Build resilient generative AI agents](#)
- [Planning for failure: How to make generative AI workloads more resilient](#)
- [Effectively building AI agents on AWS Serverless](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Observability and monitoring for agentic systems

Observability in agentic systems requires capturing not just infrastructure metrics, but also agent reasoning steps, decision paths, and workflow execution patterns. Traditional monitoring approaches must be extended to provide visibility into agent behavior, tool invocations, and multi-agent interactions.

AGENTOPS05: How do you implement comprehensive observability and monitoring for agentic systems?

Capability intent

- Agent executions produce distributed traces that capture reasoning steps, tool invocations, memory operations, and inter-agent handoffs, with trace context propagated across every service boundary.
- Behavioral baselines are continually maintained for each agent, and drift, anomalies, and performance degradation are detected automatically before they impact users.
- Agent decisions, actions, and reasoning are captured in structured, queryable logs and immutable, PII-safe audit trails that support debugging, compliance reporting, and forensic analysis.
- Workflow effectiveness is measured through a defined KPI framework covering operational, quality, efficiency, and business dimensions, and KPIs are reported to technical and business stakeholders on a regular cadence.

- Operators have workflow-specific dashboards that surface health, bottlenecks, and remediation runbooks within seconds, shortening mean time to detect and mean time to resolve for agent incidents.

Maturity levels

These levels summarize what each stage of maturity looks like for observability and monitoring of agentic systems as a whole.

Level	Name	What it looks like
1	Initial	Observability is limited to infrastructure metrics, such as AWS Lambda duration and Amazon API Gateway latency. Agent reasoning, tool invocations, and memory operations are not instrumented. Logs are unstructured and scattered across accounts. Incidents are diagnosed through manual log correlation. No behavioral baselines, agent KPIs, or immutable audit trails exist.
2	Emerging	Primary agent workflows are instrumented with distributed tracing using Amazon Bedrock AgentCore Observability or OpenTelemetry spans. Structured JSON logs flow to Amazon CloudWatch Logs with documented schemas. Basic Amazon CloudWatch dashboards and static threshold alarms exist. Initial

Level	Name	What it looks like
		log retention is configured for operational and compliance needs.
3	Defined	Telemetry schemas are standardized across agents, and end-to-end trace correlation is achieved through W3C Trace Context propagation. Behavioral baselines are collected and Amazon CloudWatch Anomaly Detection is configured for key metrics. A KPI framework covers operational, quality, efficiency, and business dimensions. Workflow-specific dashboards are linked to runbooks, and structured logs are queried through Amazon CloudWatch Logs Insights .

Level	Name	What it looks like
4	Proactive	Anomaly detection distinguishes data drift, concept drift, and performance drift, and alerts are routed by type and severity through automated response workflows. KPIs are reviewed on a regular cadence and baselines adjust automatically. Immutable audit trails use Amazon S3 Object Lock in compliance mode with PII redaction through Amazon Bedrock Guardrails sensitive information filters . Quality KPIs incorporate Amazon Bedrock AgentCore Evaluations scores, and dashboards are annotated with deployment events to correlate metric changes with configuration changes.

Level	Name	What it looks like
5	Optimized	Observability is self-healing. Anomaly detection triggers automated remediation for known patterns, and predictive KPI forecasting flags degradation before thresholds are exceeded. Continuous learning loops feed observability signals back into agent configuration, model selection, and tool design. The organization contributes to industry best practices for agentic AI observability.

Common issues to watch for

- Organizations instrument infrastructure metrics but not agent-specific spans such as reasoning iterations, tool invocations, and memory operations.
- Teams implement tracing without propagating context across agent boundaries, producing disconnected trace fragments that can't be reassembled into the workflow operators need to diagnose incidents.
- Behavioral baselines are set once at deployment and never refreshed, so legitimate behavioral evolution generates false positive alerts while gradual drift that redefines the baseline goes undetected.
- Logs are stored in mutable storage without integrity controls or personally identifiable information (PII) redaction, weakening the evidentiary value of audit trails and creating data protection exposures at the same time.
- KPIs and dashboards are defined at launch and never revisited as workflows evolve, so organizations continue to measure metrics that no longer reflect what matters while new failure modes go unmonitored.

Best practices

- [AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations](#)
- [AGENTOPS05-BP02 Monitor agent behavior patterns and detect anomalies](#)
- [AGENTOPS05-BP03 Implement structured logging and comprehensive audit trails](#)
- [AGENTOPS05-BP04 Define and track KPIs for agent workflows](#)
- [AGENTOPS05-BP05 Create workflow-specific dashboards for operational health](#)

AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations

When an agent produces an unexpected output, the investigation is only as good as the telemetry. Distributed tracing that captures the full execution path (reasoning, tool calls, memory operations, and model invocations) enables precise reconstruction of every decision and action.

Desired outcome:

- Every agent run produces a complete distributed trace covering the flow from request to response across all services and agents.
- Teams can reconstruct the exact sequence of operations for any run, enabling rapid debugging and targeted optimization.
- Real-time telemetry dashboards give operational teams continuous visibility into agent health.
- Trace data is retained on a defined policy for post-operations analysis and compliance.

Common anti-patterns:

- Instrumenting only infrastructure metrics (Lambda duration, API Gateway latency) without capturing agent-specific spans for reasoning steps, tool invocations, and memory operations.
- Implementing tracing without propagating trace context across agent boundaries, producing disconnected trace fragments that can't be correlated into end-to-end workflows.
- Capturing telemetry without standardized schemas, making it impossible to query consistently across agents or compare behavior across versions.
- Retaining traces forever because no one defined a policy, or retaining them too briefly to support quarterly trend analysis.

Benefits of establishing this best practice:

- Distributed tracing makes agent operations like decisions and actions queryable.
- Detailed telemetry provides the empirical foundation for optimization, identifying bottlenecks and validating improvements with data.
- Trace context propagation across agent boundaries makes multi-agent workflow debugging tractable.
- Standardized schemas enable cross-agent comparison and version-over-version regression analysis.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock AgentCore Observability](#) is the default telemetry service for agents on AgentCore Runtime. Its OpenTelemetry-compatible instrumentation automatically captures LLM inference calls, tool invocations, and memory operations without requiring each agent to add its own spans. For agents built on Strands Agents or custom frameworks, the agent loop itself needs instrumentation. OpenTelemetry spans wrapping each operation phase make the trace complete.

Trace context propagation separates useful traces from fragmented ones. W3C Trace Context propagation across all agent boundaries maintains continuity in distributed workflows, so a request that passes through five agents produces one trace with five spans, not five disconnected traces. Without propagation, multi-agent debugging becomes manual correlation by timestamp, which scales poorly and produces incorrect answers when concurrent requests overlap.

Standardized span schemas produce queryable data from your telemetry. Each span type needs defined fields, like model ID and token counts for inference, tool name and latency for invocations, and iteration count for reasoning. Store telemetry in [Amazon CloudWatch](#) Logs with structured JSON so dashboards and queries work against named fields. Configure sampling to capture 100% of error traces and a configurable percentage of successful traces. This balances visibility with cost, and errors are not dropped.

Implementation steps

1. **Instrument agents with OpenTelemetry spans:** Deploy on [Amazon Bedrock AgentCore Observability](#) or add manual instrumentation covering all operation phases (reasoning, tool calls, memory operations).

2. **Propagate W3C Trace Context across agent boundaries:** Carry trace context forward on every agent-to-agent, agent-to-tool, and agent-to-service call.
3. **Define standardized telemetry schemas:** Specify fields for each span type, and log in structured JSON for efficient [Amazon CloudWatch](#) Logs queries.
4. **Build end-to-end dashboards:** Visualize agent performance with drill-down to individual trace components.
5. **Set retention policies:** Balance visibility with storage cost, with different tiers for operational, compliance, and debug telemetry.

Resources

Related best practices:

- [AGENTOPS05-BP02 Monitor agent behavior patterns and detect anomalies](#)
- [AGENTOPS05-BP03 Implement structured logging and comprehensive audit trails](#)
- [AGENTOPS04-BP03 Develop fallback behavior and error handling for tool invocations](#)
- [AGENTPERF01-BP02 Implement comprehensive performance telemetry](#)
- [AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery](#)

Related documents:

- [Getting started with Amazon Bedrock AgentCore Observability](#)
- [Operationalizing agentic AI on AWS](#)
- [Observing agentic AI workloads using Amazon CloudWatch](#)
- [Observability and monitoring for serverless agentic AI on AWS](#)

Related videos:

- [AWS 2025 - AgentCore Observability: Monitor and Debug with OpenTelemetry](#)
- [AWS re:Invent 2024 - Observability for Reliable Agentic AI with Strands & OpenTelemetry \(NTA406\)](#)
- [AWS re:Invent 2024 - Build observable AI agents with Strands, AgentCore, and Datadog \(AIM233\)](#)
- [AWS 2025 - Strands Agents Observability, Evaluation, & Deployment](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore, Lab 4: Deploy to Production](#)
- [Diving Deep into Bedrock AgentCore, Observability](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS X-Ray](#)
- [Amazon CloudWatch](#)

AGENTOPS05-BP02 Monitor agent behavior patterns and detect anomalies

Static threshold alerts catch obvious breakages. Anomaly detection over behavioral baselines extends coverage to gradual shifts like a slowly rising escalation rate, a quietly increasing hallucination frequency, or tool-selection patterns that drift toward less capable options, providing early visibility into behavioral trends.

Desired outcome:

- Baseline behavior profiles are established per agent and updated continually as normal behavior evolves.
- Anomalies (unusual reasoning patterns, unexpected tool usage, performance degradation, and behavioral drift) are detected automatically and routed to the right response workflow.
- Teams receive early warning of emerging issues before they impact users.
- Behavioral changes can be traced to specific configuration updates, model updates, or input distribution shifts.

Common anti-patterns:

- Relying exclusively on static threshold-based alerting without anomaly detection, missing gradual drift that never triggers a single threshold but represents a significant cumulative change.
- Establishing behavior baselines once at deployment without updating them as normal behavior evolves, so legitimate evolution is flagged as anomalous.
- Monitoring only performance metrics without behavioral metrics (reasoning patterns, tool selection, escalation rate), missing anomalies that don't manifest as performance issues.
- Treating every anomaly with the same urgency, producing alert fatigue that causes teams to ignore genuine issues.
- Failing to distinguish data drift (input distribution shifts), concept drift (input-output relationship changes), and performance drift (output quality degradation), leading to misdirected remediation.

Benefits of establishing this best practice:

- Behavioral monitoring extends observability from infrastructure metrics to decision-making patterns, giving visibility into the aspects of agent behavior that most affect business outcomes.
- Drift detection creates a feedback signal that identifies when agents need retraining, reconfiguration, or updates to maintain alignment.
- Severity-aware routing keeps teams responsive to high-impact anomalies without drowning them in low-severity signal.
- Correlating anomalies with configuration and model changes accelerates root-cause analysis.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Establish a baseline, as anomaly detection without a baseline can produce unreliable signals. Collect agent metrics over two to four weeks, like reasoning iteration counts, tool selection frequency, escalation rates, task completion rates, and confidence score distributions, to establish a baseline for each agent. [Amazon CloudWatch](#) Anomaly Detection produces dynamic baselines that evolve automatically and detects deviations through ML-based bands, which catches drift that static thresholds miss.

Choose your metrics carefully. Performance metrics (latency, error rate) are necessary but not sufficient. Behavioral metrics, reasoning patterns, tool selection frequency, escalation rate, and

output quality distributions are where subtle drift first appears and where the anomaly that affects users most commonly occurs.

Severity-based routing helps prevent alert fatigue from eroding the system's usefulness.

- Performance anomalies trigger automated investigation
- Behavioral anomalies trigger human review
- Security-relevant anomalies trigger immediate escalation

The three queues serve different operational loops, and mixing them can produce noise and under-response. Correlate anomalies with deployment events on dashboards.

Rolling baseline updates keep the system aligned with legitimate change. As agents accumulate usage and mature, normal behavior shifts, and a baseline frozen at deployment will eventually flag every day as anomalous. The update cadence should reflect the agent's stability: weekly rolling windows work for agents under active iteration, monthly or longer for stable production agents.

Implementation steps

1. **Define behavioral metrics per agent:** Cover reasoning patterns, tool usage, escalation rates, and output quality alongside performance metrics.
2. **Collect baselines and configure [Amazon CloudWatch Anomaly Detection](#):** Use a representative observation period and configure anomaly detection bands on key behavioral metrics.
3. **Route anomalies by type and severity:** Performance anomalies to automated investigation, behavioral to human review, security to immediate escalation.
4. **Build behavioral monitoring dashboards:** Show patterns over time and correlate with configuration or model changes.
5. **Update baselines on a rolling basis:** Reflect legitimate evolution while maintaining sensitivity to genuine anomalies.

Resources

Related best practices:

- [AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations](#)
- [AGENTOPS05-BP03 Implement structured logging and comprehensive audit trails](#)

- [AGENTOPS02-BP04 Maintain feedback control loops for continuous improvement](#)
- [AGENTRELO2-BP03 Implement behavioral anomaly detection and monitoring](#)
- [AGENTSEC07-BP04 Behavioral anomaly detection and agent containment](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Launching Amazon CloudWatch generative AI observability](#)
- [Guidance for Agentic AI Operational Foundations on AWS](#)
- [Advancing AI agent governance with Boomi and AWS](#)

Related videos:

- [AWS re:Invent 2024 - Move beyond reactive: Transform cloud ops with AWS DevOps Agent \(COP362\)](#)

Related services:

- [Amazon CloudWatch](#)
- [Amazon Bedrock AgentCore](#)

AGENTOPS05-BP03 Implement structured logging and comprehensive audit trails

Free-text logs look useful until someone tries to query them at scale. Structured logs, immutable audit trails, and defined retention policies make your logs an active operational tool that provides evidence for compliance.

Desired outcome:

- All agent decisions, actions, and interactions are captured in structured, queryable logs.
- Audit trails are immutable and tamper-evident, providing a trustworthy record for regulatory and governance purposes.
- Log retention policies balance operational and compliance needs with storage cost.
- Authorized teams query log data efficiently through defined interfaces.

Common anti-patterns:

- Using unstructured free-text logging that can't be efficiently queried or parsed at scale.
- Logging only errors and exceptions without capturing successful operations, producing an incomplete picture that reduces the ability to reconstruct the full sequence.
- Storing logs in mutable storage without integrity controls, creating audit trails that could be altered and therefore can't be relied upon for compliance.
- Logging sensitive information, personally identifiable information (PII) or credentials, in agent reasoning traces, creating compliance and security risk.
- Operating without retention policies, producing unbounded log volumes that become expensive to store and difficult to search.

Benefits of establishing this best practice:

- Immutable, structured audit trails provide the evidentiary foundation for regulatory compliance and demonstrate that agents operated within authorized boundaries.
- Structured logging with efficient query interfaces turns log data from a passive record into an active operational tool, enabling rapid incident investigation and pattern extraction.
- PII redaction at write time helps prevent sensitive information from reaching log storage, reducing data protection risk.
- Tiered retention keeps compliance-relevant logs available for years while retiring short-term debug logs that would otherwise accumulate cost.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Structured logging is a format discipline that verifies that your other logging practices work correctly. Your JSON should have a standardized schema to make it queryable, including:

- Timestamp
- Trace ID
- Agent ID
- Session ID
- Operation type

- Decision rationale
- Outcome

Free-text logs require regex searches to answer simple questions, while structured logs answer them through [Amazon CloudWatch Logs](#) Insights queries against named fields. Enforce the schema, even if it is simple.

Your retention policy should depend on the purpose of the logs.

- Operational logs (30–90 days) support incident investigation and recent trend analysis.
- Compliance logs (1–7 years depending on regulatory requirements) support audits and legal discovery.
- Debug logs (7–14 days) support development and are expensive to keep beyond that.

Applying different retention policies to different log streams, rather than one policy to everything, cuts storage cost substantially without losing important log information.

PII redaction should happen before logs reach storage. [Amazon Bedrock Guardrails](#) sensitive information filters detect and redact PII at write time, which is the only reliable place to do it. Once PII is in the log, every downstream access becomes a data protection concern.

For compliance-critical logs, [Amazon S3](#) with Object Lock in Compliance mode provides immutable storage that supports regulatory requirements for tamper-evident audit trails. [AWS CloudTrail](#) captures API-level agent actions as an infrastructure complement, and [Amazon Bedrock AgentCore Observability](#) captures agent reasoning chains, tool invocations, and decision artifacts automatically for agents on AgentCore Runtime.

Establish saved query templates to reduce investigation latency. For security-focused immutable audit logs with cryptographic integrity, see [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#).

Implementation steps

1. **Define a JSON log schema:** Cover trace ID, operation type, decision rationale, and outcome as standard fields for every agent operation.
2. **Configure tiered retention:** Separate operational (30–90 days), compliance (1–7 years), and debug (7–14 days) log streams.

3. **Redact PII before write:** Integrate [Amazon Bedrock Guardrails](#) sensitive information filters into the logging pipeline.
4. **Use immutable storage for compliance logs:** Write audit trails to [Amazon S3](#) with Object Lock in Compliance mode.
5. **Create saved query templates:** Cover common operational analysis patterns so incident response doesn't start from a blank screen.

Resources

Related best practices:

- [AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations](#)
- [AGENTOPS05-BP02 Monitor agent behavior patterns and detect anomalies](#)
- [AGENTOPS04-BP02 Establish standardized tool integration protocols \(MCP, A2A\)](#)
- [AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery](#)
- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Observing agentic AI workloads using Amazon CloudWatch](#)
- [Getting started with Amazon Bedrock AgentCore Observability](#)
- [Advancing AI agent governance with Boomi and AWS](#)

Related services:

- [Amazon CloudWatch](#)
- [AWS CloudTrail](#)
- [Amazon S3](#)
- [Amazon Bedrock AgentCore](#)

AGENTOPS05-BP04 Define and track KPIs for agent workflows

Infrastructure metrics like CPU, memory, and invocation count explain whether an agent is running. However, these metrics don't determine whether an agent is actually working. Key performance indicators (KPIs) tied to business outcomes give teams and stakeholders a shared language for discussing and improving agent performance.

Desired outcome:

- Every agent workflow has a defined set of KPIs tracked continually against established baselines.
- Teams identify performance degradation early and correlate KPI changes with configuration or model updates.
- Optimization efforts are prioritized based on measurable impact rather than intuition.
- Business stakeholders have regular visibility into how agent workflows contribute to business outcomes.

Common anti-patterns:

- Tracking only infrastructure metrics (like CPU, memory, and invocation count) without agent-specific KPIs that measure business outcomes like task completion rate and user satisfaction.
- Defining KPIs at deployment and never revisiting them as business objectives evolve, measuring metrics that no longer reflect what matters.
- Collecting KPI data without establishing baselines or alerting thresholds, producing dashboards that no one monitors proactively.
- Weighting operational and business metrics equally when one matters ten times more than the other, creating dashboards that feel balanced but mislead.

Benefits of establishing this best practice:

- KPIs provide the quantitative foundation for evidence-based decisions, so teams measure change impact and prioritize improvements based on data.
- Trend tracking reveals patterns of degradation or improvement that inform continuous refinement of prompts, configurations, and integrations.
- Business outcome metrics connect agent work to value delivered, giving stakeholders regular, concrete updates instead of vague reassurance.
- Anomaly-based alerting catches gradual degradation that static thresholds miss.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

A usable KPI framework covers four dimensions rather than one.

- Operational KPIs, like task completion rate, resolution time, error rate, and escalation rate, measure whether the agent runs reliably.
- Quality KPIs, like decision accuracy, hallucination rate, and user satisfaction, measure whether its outputs are correct. [Amazon Bedrock AgentCore Evaluations](#) scores for correctness, helpfulness, safety, and tool selection accuracy, which provides automated quality signals.
- Efficiency KPIs, like tokens per task, tool invocations per task, and cost per task, measure whether the agent is economical.
- Business KPIs, like outcome achievement rate, SLA compliance, and customer satisfaction impact, measure whether the agent is worth the investment.

Skipping any dimension produces a dashboard that looks complete and can be misleading.

Baselines make KPIs useful by providing context for comparison. Establish baselines during an initial observation period (two to four weeks is usually enough), then configure [Amazon CloudWatch](#) Anomaly Detection so baselines adjust automatically as workflows mature. Set warning and critical alerting thresholds, where warnings initiate review and critical alerts dictate the need for action.

Weekly KPI reports through [Amazon Quick Suite](#) share the same dashboard with technical and business stakeholders. The same metrics serve both audiences so that everyone sees the same trajectory and conversations about investment and prioritization have shared data to ground them. Quarterly reviews verify that KPI definitions still reflect up-to-date business objectives and metrics.

Implementation steps

1. **Define a four-dimensional KPI framework:** Cover operational, quality, efficiency, and business dimensions for each agent workflow, with use-case-specific weighting.
2. **Collect KPIs through [Amazon CloudWatch](#) custom metrics:** Add dimensions for agent, workflow, and environment so the same metric can be sliced multiple ways.
3. **Establish baselines and configure anomaly detection:** Use [Amazon CloudWatch](#) Anomaly Detection with warning and critical alerting thresholds.

4. **Build weekly KPI dashboards:** Use [Amazon Quick](#) reports shared with technical and business stakeholders.
5. **Review KPI alignment quarterly:** Verify that definitions still reflect current business objectives, and retire or replace metrics that no longer apply.

Resources

Related best practices:

- [AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations](#)
- [AGENTOPS05-BP05 Create workflow-specific dashboards for operational health](#)
- [AGENTOPS02-BP04 Maintain feedback control loops for continuous improvement](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Evaluating AI agents: Real-world lessons from building agentic systems at Amazon](#)
- [Guidance for Agentic AI Operational Foundations on AWS](#)
- [From AI agent prototype to product: Lessons from building AWS DevOps Agent](#)

Related services:

- [Amazon CloudWatch](#)
- [Amazon Quick](#)
- [Amazon Bedrock AgentCore](#)

AGENTOPS05-BP05 Create workflow-specific dashboards for operational health

Generic infrastructure dashboards can hide important metrics to agentic workflows. A dashboard designed around a specific workflow's critical path, step-level latencies, and characteristic failure modes provides detail that operators need to quickly see issues and understand the root cause.

Desired outcome:

- Each critical agent workflow has a dedicated dashboard with real-time visibility into workflow health.
- Operators identify issues and quickly understand root causes.
- Dashboards are tailored to the specific characteristics of each workflow, not generic templates.
- Operational teams use these dashboards as the primary tool for workflow monitoring and incident response.

Common anti-patterns:

- Using generic infrastructure dashboards for all agent workflows, missing workflow-specific metrics like handoff success rates, reasoning iteration counts, and step-level bottlenecks.
- Building dashboards without linking to operational runbooks, forcing operators to search for remediation during incidents instead of navigating directly.
- Creating dashboards once and never updating them as workflows evolve, so metrics for steps that no longer exist stay visible while new steps go unmonitored.
- Building dashboards that require deep context to interpret, so only the original author can make sense of them.

Benefits of establishing this best practice:

- Workflow-specific dashboards expose the metrics and patterns most relevant to each workflow's operational characteristics and failure modes.
- Tailored dashboards adapt monitoring depth to each workflow's criticality, providing detail for critical workflows and overview for less critical ones.
- Embedded runbook links compress the time from detection to remediation.
- Deployment event annotations correlate metric changes with configuration changes, giving operators attribution without cross-referencing tools.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

A common layout, like top-level health summary (healthy, degraded, and critical), key metrics as time-series graphs, and recent events and alerts, means that operators learn the pattern once

and apply it to every workflow dashboard. Each workflow then adds its specific content, like the critical-path steps, their completion times, their success rates, and their queue depths.

Identifying the critical path within the dashboard. [Amazon CloudWatch](#) Contributor Insights identifies top contributors to errors and latency empirically rather than by guesswork, which is often more accurate than what the team assumes.

Workflow state visualization, the distribution of in-flight requests across steps, is a view that reveals accumulation points. A step that holds more requests than expected is either running slowly or is the gate before a downstream failure. Either way, the operator sees the problem without having to reconstruct it from separate latency and error metrics. Deployment event annotations then tie metric changes back to configuration changes, compressing root-cause investigation.

Embed runbook links to lower the time to detect and remediate issues. An operator looking at a degraded dashboard should be one click from the runbook for that failure mode. Establish a review cadence, typically quarterly, so dashboards stay aligned with workflow changes rather than drifting into obsolete representations.

Implementation steps

1. **Identify workflows that warrant dedicated dashboards:** Base the list on business impact and incident history.
2. **Design a consistent dashboard layout:** Apply the same health summary, key metrics, and recent events pattern to every workflow dashboard.
3. **Visualize the critical path:** Show step-level latency and success rates for the steps where bottlenecks typically form, using [Amazon CloudWatch](#) Contributor Insights to identify top contributors empirically.
4. **Annotate deployment events:** Correlate metric changes with configuration deployments so attribution is visible on the dashboard.
5. **Embed runbook links and review quarterly:** Link each dashboard to the runbook for common failure scenarios, and update dashboards as workflows change.

Resources

Related best practices:

- [AGENTOPS05-BP04 Define and track KPIs for agent workflows](#)

- [AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations](#)
- [AGENTOPS05-BP02 Monitor agent behavior patterns and detect anomalies](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Observing agentic AI workloads using Amazon CloudWatch](#)
- [Guidance for Agentic AI Operational Foundations on AWS](#)

Related services:

- [Amazon CloudWatch](#)
- [Amazon Bedrock AgentCore](#)

Testing, evaluation, and validation frameworks

Systematic testing and evaluation processes confirm agent quality, reliability, and alignment with business objectives through both traditional software testing and AI-specific quality assessment. Without a framework that covers every stage of the agent lifecycle, quality regressions from prompt changes, tool updates, or model updates reach users before anyone notices.

AGENTOPS06: How do you implement testing, evaluation, and validation frameworks?

Capability intent

- Agents are validated at every layer, from isolated components to entire workflows running in production-shadow mode, before changes reach users.
- Quality, safety, efficiency, and business-alignment metrics are measured continually against version-controlled benchmarks, with regressions surfaced as soon as they appear.
- Change governance is proportional to risk. Low-risk changes flow through automated gates, and high-risk changes receive SME and business-owner review.
- Evaluation datasets, prompts, and scoring rubrics are versioned and kept current as agent capabilities and use cases evolve.

- Rollback paths for prompts, tools, models, and agent versions are defined, rehearsed, and wired to the same telemetry that detects quality threshold violations.

Maturity levels

These levels summarize what each stage of maturity looks like for testing, evaluation, and validation frameworks as a whole.

Level	Name	What it looks like
1	Initial	Testing is one-time and focused on the happy path. Teams rely on manual inspection of agent outputs and infrequent exact-match assertions, missing failures that only appear in tool-use, multi-agent, or production traffic paths. No formal evaluation metrics exist beyond deployment-time smoke tests. Approvals are informal emails or chats, and rollback is a manual, untested redeploy of an older artifact.
2	Emerging	Unit and integration tests exist for the core reasoning, tool, and memory components, and run in a CI/CD pipeline. Evaluation is performed at key milestones using Amazon Bedrock model evaluation and documented acceptance criteria. Approval workflows distinguish at least two risk tiers, and rollback

Level	Name	What it looks like
		procedures are documented per agent. Coverage of edge cases, adversarial inputs, and production shadow runs is still patchy.
3	Defined	A four-layer testing pyramid, covering unit, integration, end-to-end, and shadow, is standard across agent teams, with Amazon Bedrock AgentCore Evaluations supplying built-in evaluators as standardized quality gates. Continuous evaluation tracks output quality, safety, efficiency, and business alignment over time, and risk-tiered approval workflows route changes by scope of impact. Rollback procedures are automated through pipeline triggers and exercised regularly.

Level	Name	What it looks like
4	Proactive	Shadow testing runs alongside production for every significant release, with divergences triaged through Amazon CloudWatch traces correlated to evaluation scores. Online evaluation samples live interactions continuously, with thresholds that trigger automated rollback through deployment alarms. Risk-tiered approvals include business owners and SMEs for autonomy-increasing changes, and evaluation datasets evolve automatically from incident and feedback data. Quality, safety, and business metrics are tracked across agent versions as first-class release artifacts.

Level	Name	What it looks like
5	Optimized	Testing, evaluation, and validation are an integrated, self-healing system. Custom evaluators encode organization-specific quality standards and are versioned with the agents they cover. Policy-as-code expresses risk-tier rules and approval routing, and rollback is provably correct, tested quarterly through game-day exercises. Evaluation scoring, shadow comparisons, and human feedback feed a continuous improvement loop that updates datasets, prompts, and guardrails automatically. The organization contributes patterns and evaluators back to internal communities and external forums.

Common issues to watch for

- Teams stop at deterministic unit tests and skip shadow testing on real traffic, so agents ship with unknown behavior on edge cases, adversarial inputs, and production data distributions.
- Quality is measured once at release and never again, so data drift, prompt decay, and upstream model updates silently erode output quality between releases.
- All changes flow through either a heavyweight board or a rubber-stamp review, which either bottlenecks minor tweaks or waves through autonomy increases without SME scrutiny.
- Test and evaluation datasets are created once and never refreshed, so scores stay green while real-world failure modes go undetected.

- Rollback procedures live in a runbook but are never rehearsed, so the first real attempt during an incident discovers broken artifact stores, missing permissions, or prompt and tool version mismatches.

Best practices

- [AGENTOPS06-BP01 Design multi-layered testing frameworks](#)
- [AGENTOPS06-BP02 Evaluate and track ongoing agent performance](#)
- [AGENTOPS06-BP03 Establish SME-driven validation and business approval workflows](#)

AGENTOPS06-BP01 Design multi-layered testing frameworks

Traditional software testing, like exact-match assertions and green-or-red unit tests, can miss important failure modes in agentic systems. A testing pyramid that covers unit, integration, end-to-end tests, and shadow layers helps teams catch behavioral regressions before they reach users.

Desired outcome:

- Agent systems are covered by a testing pyramid that includes unit tests, integration tests, end-to-end tests, and shadow tests in production environments.
- Automated testing pipelines run on every code and configuration change, providing rapid feedback on regressions.
- Test coverage metrics are tracked and maintained above defined thresholds for all agent capabilities.
- Tests use semantic quality assessment rather than exact-match comparison, so non-deterministic outputs don't break the suite.

Common anti-patterns:

- Testing only the happy path without covering edge cases, error conditions, and adversarial inputs.
- Relying exclusively on unit tests without integration and end-to-end tests, missing failures that only emerge when components interact with real tools and services.
- Treating agent testing as equivalent to traditional software testing without accounting for non-deterministic LLM outputs, using exact string matching instead of semantic equivalence checks.

- Running tests only in isolated environments without shadow testing in production, missing environment-specific behaviors that only manifest with real data and traffic patterns.
- Failing to maintain test datasets as capabilities evolve, so tests become stale and lose regression-detection value.

Benefits of establishing this best practice:

- A thorough testing framework provides the empirical evidence needed to validate each behavioral iteration, enabling confident deployment.
- Standardized testing procedures help validate every change consistently, regardless of who made it or how urgent the timeline.
- Semantic evaluation accepts legitimate output variation while still catching regressions.
- Shadow testing validates behavioral changes against real traffic without exposing users to the new version.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Four layers cover the testing surface for most agent systems.

Unit tests, the base layer, test individual components in isolation: prompt templates, tool invocation logic, memory retrieval, decision routing. LLM responses can be mocked where determinism is needed, so unit tests stay fast and reproducible.

Integration tests, the second layer, validate agent-tool and agent-to-agent interactions in a staging environment with real endpoints, which is where many of the interesting failures emerge.

End-to-end tests, the third layer, validate complete workflows, and this is where semantic evaluation matters more than exact matching. [Amazon Bedrock Evaluations](#) and [Amazon Bedrock AgentCore Evaluations](#) handle the semantic quality assessment that end-to-end tests need. AgentCore Evaluations' 13 built-in evaluators provide standardized quality gates in CI/CD pipelines (correctness, helpfulness, safety, and tool selection accuracy), so regressions in output quality are detectable without requiring bit-exact comparison. Custom evaluators cover business-specific requirements.

Shadow tests, the top layer, run new versions in parallel with production on real traffic using traffic mirroring, comparing outputs without serving the new version's responses. This catches

environment-specific behavior that staging can't reproduce. The cost is the infrastructure to run parallel inferences, and the value is catching issues before users ever encounter them. For teams developing agents with Kiro, hooks can trigger test runs on file save and before deployment.

Integrate automated testing into CI/CD pipelines so every layer blocks deployment on failure. Maintain test datasets with versioning, and review them regularly to add new use cases and failure modes discovered in production. The pyramid gets stronger over time only if the suite grows with the system.

Implementation steps

1. **Define the four testing layers:** Scope, tooling, and success criteria for unit, integration, end-to-end, and shadow tests.
2. **Implement unit and integration tests:** Mock dependencies at the unit layer. Use real staging endpoints for integration tests.
3. **Create end-to-end scenarios with semantic evaluation:** Use [Amazon Bedrock AgentCore Evaluations](#) for quality assessment rather than exact-match assertions.
4. **Add shadow testing with traffic mirroring:** Validate behavioral changes against real-world inputs without exposing users.
5. **Integrate tests into CI/CD:** Run the full suite on every commit and block deployment on failures.

Resources

Related best practices:

- [AGENTOPS06-BP02 Evaluate and track ongoing agent performance](#)
- [AGENTOPS06-BP03 Establish SME-driven validation and business approval workflows](#)
- [AGENTOPS03-BP02 Implement CI/CD pipelines tailored to agentic system deployment \(AgentOps\)](#)
- [AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Amazon Bedrock AgentCore Evaluations](#)

- [Evaluating AI agents: Real-world lessons from building agentic systems at Amazon](#)
- [LLM-as-a-judge on Amazon Bedrock Model Evaluation](#)
- [Evaluate models in Amazon Bedrock](#)
- [Evaluating AI agents for production: A practical guide to Strands Evals](#)
- [From AI agent prototype to product: Lessons from building AWS DevOps Agent](#)
- [Kiro Hooks](#)

Related videos:

- [AWS 2025 - Strands Agents Observability, Evaluation, & Deployment](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples, Evaluations tutorials](#)
- [GitHub: awslabs/amazon-bedrock-agent-samples, RAGAS evaluation](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore, Lab 5: Evaluate Agent Performance](#)
- [Diving Deep into Bedrock AgentCore, Evaluations](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTOPS06-BP02 Evaluate and track ongoing agent performance

Pre-deployment evaluation validates that an agent is ready to ship. Post-deployment evaluation validates that it still works. Without continuous assessment, gradual quality degradation from data drift, model updates, and shifting user patterns goes unnoticed until it is expensive to fix.

Desired outcome:

- Agent performance is continually evaluated against defined quality benchmarks.
- Automated pipelines detect degradation in output quality, reasoning accuracy, and business outcome alignment.
- Teams have clear visibility into performance trends over time and can correlate quality changes with specific configuration, model, or data updates.
- Evaluation results drive prioritized improvement actions and provide objective evidence for stakeholder reporting.

Common anti-patterns:

- Evaluating agent performance only at deployment time without continuous post-deployment assessment, missing gradual degradation from data drift, model updates, or changing user patterns.
- Relying solely on automated metrics without periodic human evaluation, missing quality dimensions that automated metrics can't fully capture (like nuance, appropriateness, and business context alignment).
- Using generic evaluation criteria across all agents without tailoring metrics to each agent's specific use case and business objectives, producing evaluation results that don't reflect actual value.
- Treating evaluation as separate from operations rather than integrating it into the operational workflow, creating evaluation debt that accumulates over time.

Benefits of establishing this best practice:

- Continuous evaluation provides an empirical foundation for evidence-based improvement, identifying which agents need attention and which changes produce measurable gains.
- Performance trend tracking reveals patterns that inform systematic improvement, turning evaluation data into practical insights.
- Multi-dimensional scoring catches quality issues that a single metric would miss.
- Correlation between quality shifts and configuration changes compresses root-cause analysis.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock AgentCore Evaluations](#) is an evaluation service for continuous assessment. Its on-demand mode runs benchmarks during development, and its online mode samples and evaluates live interactions in production without requiring manual triggers. Thirteen built-in evaluators cover correctness, helpfulness, safety, and tool selection accuracy, with custom evaluators available for business-specific requirements. [Amazon Bedrock Evaluations](#) supplements this with model-level assessment, and periodic human evaluation covers the dimensions automated metrics miss.

Evaluation frameworks need multiple dimensions because a single metric misses too much. For example:

- Output quality (relevance, accuracy, coherence) measures whether responses are good.
- Safety (hallucination rate, toxicity, guardrail adherence) measures whether responses are safe.
- Efficiency (task completion rate, tool invocation success) measures whether the agent is economical.
- Business alignment (outcome achievement, user satisfaction, SLA compliance) measures whether the agent delivers value.

Weighting depends on the use case. For instance, a customer-support agent might weigh satisfaction higher than efficiency, while an internal automation agent might weigh efficiency higher than relevance. Generic weighting produces generic results.

Dashboards that show evaluation scores over time make degradation visible before it becomes an incident. Alerting on threshold violations and on persistent negative trends, as opposed to single-point dips, catches the slow-moving problems that are hardest to diagnose after the fact. Correlate evaluation shifts with configuration and model changes so attribution is fast when a metric moves.

LLM-as-a-Judge patterns can use multiple evaluator prompts covering different quality dimensions to produce a composite score that is more reliable than any single prompt. Periodic human review validates the automated scores and catches the blind spots.

Implementation steps

1. **Configure [Amazon Bedrock AgentCore Evaluations](#):** Use on-demand mode for development benchmarking and online mode for continuous production monitoring.
2. **Define a multi-dimensional evaluation framework:** Apply use-case-specific weighting across quality, safety, efficiency, and business alignment.

3. **Implement LLM-as-judge patterns:** Use multiple evaluator prompts and supplement with periodic human evaluation.
4. **Build evaluation dashboards:** Show trends over time with alerting for threshold violations and persistent negative trends.
5. **Correlate evaluation results with change events:** Tag deployments, configuration updates, and model changes so quality shifts can be attributed quickly.

Resources

Related best practices:

- [AGENTOPS06-BP01 Design multi-layered testing frameworks](#)
- [AGENTOPS06-BP03 Establish SME-driven validation and business approval workflows](#)
- [AGENTOPS02-BP04 Maintain feedback control loops for continuous improvement](#)
- [AGENTOPS05-BP04 Define and track KPIs for agent workflows](#)
- [AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads](#)

Related documents:

- [Amazon Bedrock AgentCore Evaluations](#)
- [Evaluate models in Amazon Bedrock](#)
- [Operationalizing agentic AI on AWS](#)
- [Build reliable AI agents with Amazon Bedrock AgentCore Evaluations](#)
- [LLM-as-a-judge on Amazon Bedrock Model Evaluation](#)
- [Evaluating AI agents for production: A practical guide to Strands Evals](#)
- [From AI agent prototype to product: Lessons from building AWS DevOps Agent](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore, Lab 5: Evaluate Agent Performance](#)
- [Diving Deep into Bedrock AgentCore, Evaluations](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTOPS06-BP03 Establish SME-driven validation and business approval workflows

Heavy approval processes slow routine work and get bypassed in emergencies. On the other hand, a lack of approval processes can produce unforeseen incidents. To keep your teams moving while protecting against the changes that actually warrant scrutiny, implement risk-tiered validation (light for minor changes, thorough for autonomy increases).

Desired outcome:

- Significant agent changes pass through documented validation and approval workflows before reaching production.
- Validation checkpoints verify that changes meet quality thresholds, maintain behavioral alignment, and comply with operational boundaries.
- Rollback procedures are defined and tested for every change type.
- Approval burden scales with change risk, not uniformly across all changes.

Common anti-patterns:

- Applying the same lightweight approval process to all changes regardless of risk, treating a minor prompt wording adjustment the same as a change that increases agent autonomy or adds new tool access.
- Implementing approval workflows that require human sign-off for every change without risk-based tiering, creating bottlenecks that slow iteration and incentivize bypass.
- Defining validation checkpoints without specifying the criteria that must be met to pass, leaving approvers without objective standards and producing inconsistent decisions.
- Failing to test rollback procedures before they are needed, discovering that rollback is broken only when an incident requires rapid recovery.
- Treating validation as a one-time deployment gate rather than a continuous process, missing quality degradation after deployment.

Benefits of establishing this best practice:

- Risk-tiered approval workflows help route changes with significant potential impact to appropriate human scrutiny, while low-risk changes proceed with minimal friction.
- Documented validation and approval create an auditable record of every change decision for compliance purposes.
- Tested rollback procedures compress incident response time when validated changes still produce unexpected outcomes.
- Automated validation gates catch regressions before human approvers ever see the change.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Risk tiering creates a workable approval process.

1. Tier 1, low risk, covers minor prompt wording changes, logging configuration adjustments, and similar edits that can't materially alter agent behavior. These require automated validation only.
2. Tier 2, medium risk, covers new tool integrations, prompt structural changes, and model parameter adjustments. These require automated validation plus peer review.
3. Tier 3, high risk, covers autonomy level increases, new tool categories, model changes, and guardrail modifications. These require automated validation plus multi-stakeholder approval including technical lead and business owner.

Automated validation should run before any human approval. [Amazon Bedrock AgentCore Evaluations](#) score thresholds, behavioral regression tests, security scans, and performance benchmarks all gate promotion. A change that fails automated validation never consumes human review time, as the team stays focused on the decisions that genuinely require judgment.

Approval routing needs to handle timeouts. A change waiting on an unavailable approver for multiple days risks being bypassed or dropped. Timeout escalation, either through automatic approval for low-risk changes or escalation to a backup approver for higher-risk ones, keeps the process moving.

Rollback is a recovery path for changes that passed validation and still produced unexpected outcomes. Automated rollback triggered by post-deployment quality threshold violations is the default, while manual rollback remains available for edge cases. For tiered human oversight

patterns in reliability contexts, see [AGENTREL02-BP05 Establish tiered human oversight and approval workflows](#).

Implementation steps

1. **Define a risk-tiered change classification:** Spell out the criteria for Tier 1, Tier 2, and Tier 3 so changes are classified consistently.
2. **Define automated validation checkpoints:** Include evaluation score thresholds from [Amazon Bedrock AgentCore Evaluations](#), regression tests, security scans, and performance benchmarks.
3. **Implement approval workflows:** Route changes by tier with timeout escalation for non-responsive approvers.
4. **Automate rollback on quality threshold exceedance:** Wire post-deployment quality metrics to revert workflows.
5. **Test rollback procedures quarterly:** Document results and update procedures as the runtime evolves.

Resources

Related best practices:

- [AGENTOPS06-BP01 Design multi-layered testing frameworks](#)
- [AGENTOPS06-BP02 Evaluate and track ongoing agent performance](#)
- [AGENTOPS03-BP02 Implement CI/CD pipelines tailored to agentic system deployment \(AgentOps\)](#)
- [AGENTREL02-BP05 Establish tiered human oversight and approval workflows](#)
- [AGENTSEC04-BP02 Human-in-the-loop for critical decisions](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Operationalizing agentic AI, Part 1: A stakeholder's guide](#)
- [Preparing your business for agentic AI](#)
- [Advancing AI agent governance with Boomi and AWS: A unified approach to observability and compliance](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Operational recovery and consumption monitoring

Agentic systems must handle operational failures gracefully and help prevent runaway behavior that could affect business operations. Recovery mechanisms need to balance automation with appropriate escalation, and consumption monitoring must detect agents that exceed expected operational boundaries.

AGENTOPS07: How do you establish operational recovery and consumption monitoring?

Capability intent

- Agent failures trigger automated cutoffs, fallback chains, and recovery workflows that restore service within defined recovery time objectives.
- Operational knowledge about agent behavior, failure modes, and resolutions is captured centrally and remains searchable as people and systems change.
- Every agent change passes through a documented workflow that validates both technical correctness and continued alignment with business processes, policies, and regulatory obligations.
- Human operators have tested, rehearsed runbooks they can execute without the agent infrastructure, so business operations continue through worst-case agent outages.
- Recovery mechanisms, knowledge bases, and emergency procedures are exercised regularly, and findings feed back into improvements to the agent infrastructure and to the runbooks themselves.

Maturity levels

These levels summarize what each stage of maturity looks like for operational recovery and consumption monitoring as a whole.

Level	Name	What it looks like
1	Initial	Agent failures are handled reactively by whoever notices first. Recovery procedures are not documented and runbooks for worst-case scenarios are missing. Operational knowledge lives with individuals, and change management treats agent updates as generic code changes without considering business alignment.
2	Emerging	Retries and basic fallbacks are implemented for the most important dependencies. Post-incident reviews are held for significant incidents and findings are filed, although they are not systematically propagated into runbooks. An initial set of break-glass procedures is documented for the highest-impact business processes, and change management explicitly covers agent prompts, tools, and foundation models .
3	Defined	Automatic cutoffs, fallback chains, and recovery workflows cover every external dependency and are monitored in Amazon CloudWatch . Operational

Level	Name	What it looks like
		knowledge is captured in a centralized searchable system such as Amazon Bedrock Knowledge Bases , with post-incident reviews feeding entries directly. Change management classifies agent changes by technical scope and business impact and routes approvals accordingly, and break-glass runbooks are walked through in tabletop exercises.
4	Proactive	Recovery time objectives are defined for each failure scenario, tracked in dashboards, and validated through regular chaos engineering exercises. Business-alignment reviews run on a defined cadence, and agent-to-business-process mappings generate automated notifications through Amazon EventBridge when processes or policies change. Break-glass drills execute end-to-end in non-production environments, runbook updates are tied to drill findings, and operational knowledge has explicit owners who keep entries current.

Level	Name	What it looks like
5	Optimized	Agent recovery is self-healing and continuously tuned based on production telemetry , with automatic cutoffs, model fallbacks, and multi-agent degradation modes all adjusted against measured outcomes. Operational knowledge is queryable in natural language through internal assistants built on Amazon Bedrock Agents and is continuously maintained as part of normal operations. Change management feedback loops visibly drive infrastructure and business-process improvements. Break-glass readiness is assessed continuously through instrumented drills, and lessons from incidents shape both automated recovery and manual fallback procedures in the next iteration.

Common issues to watch for

- Recovery logic is implemented per component but is never tested in combination, so cascading failures that span multiple components are discovered only during real incidents when business impact is already significant.
- Post-incident reviews produce documents that are filed but never translated into runbook or platform changes, so the organization pays the cost of learning from each incident without compounding the value.

- Agent changes are approved on purely technical criteria while the business processes they support evolve separately, creating silent drift between agent behavior and current business rules.
- Break-glass runbooks and contact lists are created once and not maintained, so by the time they are needed they reference retired systems, decommissioned tools, or people who have moved on.
- Emergency procedures are stored in systems that depend on the same agent infrastructure they are meant to work around, leaving operators without documentation when it is needed.

Best practices

- [AGENTOPS07-BP01 Implement automated response and recovery mechanisms](#)
- [AGENTOPS07-BP02 Establish operational knowledge management systems](#)
- [AGENTOPS07-BP03 Augment change management to accommodate technical improvements and business requirements](#)
- [AGENTOPS07-BP04 Implement break-glass operational runbooks](#)

AGENTOPS07-BP01 Implement automated response and recovery mechanisms

Agents that recover from failure without human intervention keep service running and give the team the failure data they need to help prevent recurrence. Manual-only recovery scales poorly and can turn routine degradations into incidents.

Desired outcome:

- Agent systems detect and recover from common failure scenarios automatically, maintaining service availability and user experience continuity.
- Automatic cutoffs help prevent cascading failures from propagating across the agent environment.
- Fallback strategies keep agents degrading gracefully rather than failing completely when primary capabilities are unavailable.
- Recovery time objectives are defined, met, and tested regularly.

Common anti-patterns:

- Implementing retry logic without automatic cutoffs, causing agents to repeatedly invoke failing services and amplifying load on degraded systems rather than failing fast.
- Designing fallback strategies that silently degrade quality without notifying users, producing a poor experience where users receive low-quality responses without understanding why.
- Failing to define recovery time objectives for different failure scenarios, making it impossible to assess whether recovery mechanisms meet operational requirements.
- Implementing recovery mechanisms that work in isolation but fail in combination, missing failure scenarios where multiple components degrade simultaneously.
- Never testing recovery procedures under realistic failure conditions, discovering problems only during actual production incidents.

Benefits of establishing this best practice:

- Automated recovery captures detailed failure data that drives systematic improvement of agent resilience, letting teams address root causes rather than repeatedly responding to the same incidents.
- Self-healing capabilities adapt to different failure contexts, transient tool unavailability, complete service outages, and model degradation, with recovery strategies proportional to severity.
- Automatic cutoffs break cascading failures at the source rather than allowing them to propagate.
- Chaos engineering exercises validate that recovery works under realistic conditions, not just in theory.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Establish automatic cutoffs for every external dependency. Store state for the cutoff (healthy, degraded, and open) in a fast data store where every agent can read it. Thresholds depend on each dependency's reliability characteristics. Set an error rate threshold (for example, 50% errors in a 60-second window), a timeout threshold (for example, 5 consecutive timeouts), and a recovery probe interval (for example, attempt recovery every 30 seconds). Emit [Amazon CloudWatch](#) metrics on state transitions so cutoff health becomes visible across the environment.

Fallback strategies need to be designed for each capability, not copied from a template. Tool failures get fallback chains: alternative tools with equivalent capabilities, then graceful degradation, and then manual escalation. LLM inference failures get model fallback chains that

route to alternative models (Claude 3.5 Sonnet to Claude 3 Haiku, for example) when the primary is unavailable. Multi-agent coordination failures get single-agent fallback modes that handle tasks with reduced capability rather than failing completely. Each fallback should notify users when quality is degraded, not silently return a worse answer.

[AWS Step Functions](#) or equivalent durable workflow orchestration handles recovery workflows with built-in error handling, retry logic, and compensating transactions. Health check endpoints for each agent verify dependency availability and report overall health status.

Monitoring actual recovery times against objectives tells the team whether the mechanisms actually meet operational requirements. Quarterly chaos engineering exercises validate that recovery works under realistic conditions rather than just in the happy-path scenarios the original design anticipated. For reliability-focused automated recovery with classify-route-escalate patterns, see [AGENTREL07-BP02 Enable automatic recovery from agent execution failures](#).

Implementation steps

1. **Implement automatic cutoffs for every external dependency:** Define thresholds for error rate, timeout count, and recovery probing per dependency.
2. **Design fallback chains per agent capability:** Specify alternative tools, models, and degraded-mode operations, and notify users when quality is degraded.
3. **Build durable recovery workflows:** Use [AWS Step Functions](#) or equivalent with error handling, retry logic, and compensating transactions.
4. **Configure health check endpoints:** Verify dependency availability and report overall health status for each agent.
5. **Define RTOs per failure scenario:** Monitor actual recovery times against objectives in [Amazon CloudWatch](#).
6. **Run quarterly chaos engineering exercises:** Inject failures in non-production environments to validate recovery mechanisms under realistic conditions.

Resources

Related best practices:

- [AGENTOPS07-BP03 Augment change management to accommodate technical improvements and business requirements](#)
- [AGENTOPS04-BP03 Develop fallback behavior and error handling for tool invocations](#)
- [AGENTOPS05-BP02 Monitor agent behavior patterns and detect anomalies](#)
- [AGENTREL07-BP02 Enable automatic recovery from agent execution failures](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Build resilient generative AI agents](#)
- [Agentic AI in the Well-Architected Generative AI Lens](#)
- [From AI agent prototype to product: Lessons from building AWS DevOps Agent](#)
- [Introducing Amazon Bedrock AgentCore: Securely deploy and operate AI agents at any scale](#)

Related services:

- [AWS Step Functions](#)
- [Amazon CloudWatch](#)
- [AWS Lambda](#)
- [Amazon Bedrock AgentCore](#)

AGENTOPS07-BP02 Establish operational knowledge management systems

Teams that capture what they learn from incidents build institutional memory that survives personnel changes. Teams that don't lose the insight the moment the person who had it leaves, and pay the same lesson twice.

Desired outcome:

- Operational knowledge about agent behavior, failure modes, and resolution procedures is captured systematically and accessible to all team members.
- Post-incident reviews consistently produce practical insights incorporated into runbooks and operational procedures.
- Institutional knowledge survives personnel changes, enabling new team members to become effective quickly.
- Knowledge about successful interventions is captured alongside failure modes, so what works is remembered as reliably as what failed.

Common anti-patterns:

- Relying on knowledge held by individual team members rather than documented operational knowledge, creating single points of failure when people leave.

- Conducting post-incident reviews that produce reports but don't result in practical changes to runbooks or procedures.
- Storing operational knowledge in team-specific repositories inaccessible to other teams, reducing the risk of sharing and creating duplicate effort.
- Treating knowledge management as a documentation exercise rather than an active operational practice, producing documents that are created once and never updated.
- Failing to capture knowledge about successful interventions alongside failure modes, missing the opportunity to document what works and why.

Benefits of establishing this best practice:

- A systematic knowledge management system captures individual operational experience as organizational learning, making the whole team more effective over time.
- Documented operational knowledge enables consistent responses to common scenarios, reducing variability in how team members handle similar situations.
- Semantic search exposes relevant knowledge even when users don't know exact document names or categories.
- Quarterly audits keep the knowledge base current as systems and procedures evolve.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

A centralized knowledge repository is the starting point, but the search experience decides whether anyone uses it. [Amazon Bedrock Knowledge Bases](#) provides semantic search over operational documentation, so natural-language queries return relevant entries even when users don't know the exact title. Structure the knowledge base with categories for agent behavioral patterns, common failure modes and resolutions, operational procedures, and post-incident learnings. Amazon Bedrock retrieval-augmented generation turns the knowledge base into something queryable through natural language rather than keyword matching.

Structured, post-incident reviews that capture timeline, root cause, resolution steps, and preventive measures convert each significant incident into durable knowledge. Tie review outputs directly to knowledge base entries so that a new failure mode gets documented the same week it was diagnosed.

Quarterly audits validate accuracy and completeness. Agent behaviors change, services evolve, and procedures that worked last year may no longer apply. Without periodic validation, the knowledge

base slowly becomes less trustworthy. Consider building an internal operational assistant using [Amazon Bedrock Agents](#) that team members can query for guidance on common scenarios. This is especially valuable for onboarding and for incident response when timely guidance matters more than document discovery.

Implementation steps

1. **Deploy a centralized knowledge repository:** Use [Amazon Bedrock Knowledge Bases](#) for semantic search over operational documentation.
2. **Define categories and templates:** Cover behavioral patterns, failure modes, procedures, and post-incident learnings with consistent structure.
3. **Establish a post-incident review process:** Use structured templates that feed directly into the knowledge base so learning captured during review lands as durable knowledge.
4. **Implement contribution workflows:** Make it straightforward for team members to add and update entries without heavy process overhead.
5. **Audit quarterly:** Review accuracy, completeness, and relevance, and retire outdated entries.

Resources

Related best practices:

- [AGENTOPS07-BP03 Augment change management to accommodate technical improvements and business requirements](#)
- [AGENTOPS05-BP03 Implement structured logging and comprehensive audit trails](#)
- [AGENTOPS02-BP04 Maintain feedback control loops for continuous improvement](#)
- [AGENTCOST07-BP03 Create systematic optimization feedback loops for continuous improvement](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Guidance for Agentic AI Operational Foundations on AWS](#)
- [Preparing your business for agentic AI](#)
- [Introducing Amazon Bedrock AgentCore: Securely deploy and operate AI agents at any scale](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon S3](#)

AGENTOPS07-BP03 Augment change management to accommodate technical improvements and business requirements

A change management process built only for technical changes doesn't account for how agents actually evolve. Prompt tweaks, tool additions, and model upgrades all carry business implications that pure technical review doesn't catch. A process that engages both technical and business stakeholders, proportional to change scope, keeps agents aligned with the objectives they exist to serve.

Desired outcome:

- Every agent change, technical improvement or business requirement follows a documented change management process.
- Technical and business stakeholders are engaged appropriately based on change scope and impact.
- The business justification for each change is documented and traceable, so agent evolution is purposeful.
- Agents evolve in sync with organizational changes rather than drifting out of alignment.

Common anti-patterns:

- Managing agent changes through purely technical change management processes without business stakeholder involvement, allowing agents to drift out of alignment with business objectives.
- Treating all agent changes as technical changes without assessing business impact, missing changes that affect business processes, customer experience, or compliance requirements.
- Implementing change management processes so heavyweight that teams bypass them for urgent changes, creating an informal shadow process that lacks governance and traceability.
- Failing to synchronize agent changes with broader organizational changes (like process updates, policy changes, and regulatory updates), causing agents to operate based on outdated business rules.

Benefits of establishing this best practice:

- Documented change management with business justification creates an auditable record of purposeful, governed agent evolution.
- Change management captures business justification and impact assessment, creating a feedback loop that informs future prioritization.
- Two-dimensional classification (technical scope, business impact) helps the right stakeholders engage with the right changes rather than everyone reviewing everything.
- Synchronization with organizational changes helps prevent agents from operating under outdated business rules.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Classifying changes along two dimensions, technical scope and business impact, helps keep your processes proportional.

Technical scope captures what kind of change it is, like prompt update, tool change, model change, or architecture change. Business impact captures what it affects, like none, minor adjustment, significant process change, or compliance-affecting. The combination determines required approval workflows, documentation depth, and testing rigor. A prompt wording tweak with no business impact moves through the process quickly, while a new tool integration with compliance implications triggers the full review.

Business alignment reviews help you catch slow configuration drift. Agent capabilities that worked when the business process was X may not work when the business process is Y, and if no one periodically validates the alignment, the drift accumulates unnoticed. A periodic review, for example quarterly, validates whether agent capabilities remain aligned with current business processes, policies, and regulatory requirements. Establish a review mechanism where an agent-to-business-process mapping maintained in the portfolio catalog, with notifications routed to dependent agent owners when business processes are updated.

Tracking change volume, approval cycle times, and alignment metrics keeps the process itself under observation for continual improvement. Processes that take too long or catch too few problems should be reviewed and updated as needed. Monitoring the metrics validates the current tiering.

Implementation steps

1. **Define a change classification matrix:** Map technical scope and business impact to required approvals and documentation.
2. **Implement change request workflows:** Use structured templates capturing both technical and business justification.
3. **Establish periodic business alignment reviews:** Validate that agent capabilities match current business processes, policies, and regulatory requirements.
4. **Maintain agent-to-business-process mappings:** Configure notifications when processes are updated so dependent agents can be reviewed.
5. **Track change management metrics:** Monitor change volume, approval cycle times, and alignment measures to keep the process proportional.

Resources

Related best practices:

- [AGENTOPS07-BP01 Implement automated response and recovery mechanisms](#)
- [AGENTOPS07-BP02 Establish operational knowledge management systems](#)
- [AGENTOPS06-BP03 Establish SME-driven validation and business approval workflows](#)
- [AGENTOPS03-BP01 Define an agent lifecycle with clear SME ownership, testing, and governance](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Evolving software delivery for agentic AI](#)
- [Operationalizing agentic AI, Part 1: A stakeholder's guide](#)
- [Advancing AI agent governance with Boomi and AWS: A unified approach to observability and compliance](#)

Related services:

- [Amazon CloudWatch](#)
- [AWS Step Functions](#)

AGENTOPS07-BP04 Implement break-glass operational runbooks

Write and test emergency runbooks before they are needed. Rehearsed manual fallback procedures, accessible escalation paths, and current contact information turn a complete agent failure into a brief manual period rather than a prolonged outage.

Desired outcome:

- When agents fail completely or behave unexpectedly, human operators execute well-documented manual procedures that maintain business continuity.
- Break-glass runbooks are tested regularly, and operators are trained and confident in the manual procedures.
- Escalation paths and contact information stay current.
- Emergency response times meet defined objectives because procedures are documented, accessible, and practiced.

Common anti-patterns:

- Assuming agent systems will always be available and not documenting manual fallback procedures, leaving operators without guidance when agents fail during critical business operations.
- Creating break-glass runbooks once and never testing or updating them, resulting in procedures that reference outdated systems, incorrect contacts, or steps that no longer work.
- Storing emergency procedures in locations inaccessible during the outage scenarios they are designed to address (for example, in systems that depend on the failing agent infrastructure).
- Designing manual fallback processes that require specialized knowledge held by only one or two team members, creating single points of failure in emergency response.
- Failing to define clear triggers for when break-glass procedures should be activated, causing delays as operators debate whether the situation warrants manual intervention.

Benefits of establishing this best practice:

- Documented break-glass procedures support consistent emergency responses regardless of who is on call.
- Break-glass runbooks formalize the transition from automated agent operations to human-driven processes, keeping business continuity intact when agent systems can't function.

- Regular drills keep procedures current and operators ready, so response quality doesn't depend on which person happens to be on call.
- Clear activation triggers remove hesitation that delays response.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Inventory your system by identifying every critical business process that depends on agent systems and assess the impact of agent unavailability for each. The output is a ranked list of processes that need break-glass coverage, from brief inconveniences to service-stopping issues. Processes at the top of the list get runbooks first.

Each runbook should cover trigger conditions, step-by-step manual execution instructions, required access credentials, escalation contacts with primary and backup personnel, expected completion times, and criteria for returning to automated operations. The most common failure mode, and the most dangerous, is runbooks stored in systems that depend on the very infrastructure they are designed to work around. An emergency runbook stored in a wiki that depends on the same agent infrastructure is a runbook you can't read during the outage. Store runbooks in a highly available, agent-independent location, with offline copies accessible to on-call operators.

Single-person knowledge is the other common single point of failure. Designing manual procedures that only one or two people can execute produces a plan that collapses when those people are unavailable. Broaden the knowledge base through tabletop exercises, documentation that doesn't assume expertise, and regular cross-training.

Activation triggers remove hesitation from response. Clear conditions, for example, "agent error rate exceeds 50% for 15 minutes" or "complete agent infrastructure unavailability for 5 minutes", tell operators when to switch to manual procedures without waiting for judgment calls under pressure. Automated alerts that explicitly name the trigger conditions they have met make the decision obvious.

Testing keeps runbooks alive. Tabletop exercises quarterly, operators walking through runbook steps without executing them, catch outdated references and missing steps. Full drills semi-annually, operators actually executing manual procedures in a non-production environment, catch everything tabletop exercises miss. Drill results become input for runbook revisions, not just training feedback.

Implementation steps

1. **Inventory critical business processes:** Assess the impact of agent unavailability for each to produce a ranked list.
2. **Document manual fallback procedures:** Cover each critical process assuming no agent system availability, with step-by-step instructions.
3. **Establish escalation paths:** Include primary and backup contacts, with a process for keeping contact information current.
4. **Store runbooks independently:** Use a highly available, agent-independent location with offline copies accessible to on-call operators.
5. **Define clear activation triggers:** Configure automated alerts that notify operators when trigger conditions are met.
6. **Conduct exercises on a cadence:** Run tabletop exercises quarterly and full drills semi-annually, and update procedures based on findings.

Resources

Related best practices:

- [AGENTOPS07-BP01 Implement automated response and recovery mechanisms](#)
- [AGENTOPS07-BP02 Establish operational knowledge management systems](#)
- [AGENTOPS07-BP03 Augment change management to accommodate technical improvements and business requirements](#)
- [AGENTOPS04-BP03 Develop fallback behavior and error handling for tool invocations](#)
- [AGENTREL07-BP02 Enable automatic recovery from agent execution failures](#)

Related documents:

- [Operationalizing agentic AI on AWS](#)
- [Build resilient generative AI agents](#)
- [Agentic AI in the Well-Architected Generative AI Lens](#)
- [From AI agent prototype to product: Lessons from building AWS DevOps Agent](#)

Related services:

- [Amazon S3](#)
- [Amazon CloudWatch](#)

Security

The security portion of this lens provides security best practices and questions for organizations implementing autonomous AI agents and agentic systems on AWS. This focuses on securing AI agents, protecting against prompt injection, maintaining data privacy, and maintaining control over autonomous operations.

Capabilities

- [Secure agent memory and state](#)
- [Secure agent tool usage](#)
- [Agent identity and permission management](#)
- [Agent goal alignment and manipulation prevention](#)
- [Agent observability and non-repudiation](#)
- [Secure multi-agent orchestration](#)
- [Human oversight protection and agent containment](#)
- [Secure agent inputs and outputs](#)
- [Agent vulnerability scanning and penetration testing](#)

Design principles

In addition to the lens-level design principles, the security best practices in this lens are represented by at least one of the following principles:

- **Treat every input as untrusted and every output as potentially harmful:** Multi-layer validation, prompt injection defenses, and output filtering apply to user inputs, memory reads, tool I/O, and agent-to-agent messages alike. The agent itself is not a trust boundary.
- **Give each agent its own identity and the minimum privileges to do its job:** Per-agent authentication, dynamic permission boundaries, and scoped credentials per session limit what any single agent or compromised session can reach.
- **Partition memory, tools, and channels along trust boundaries:** Sessions, users, tenants, and agents get separate namespaces, integrity checks on stored state, and authenticated and encrypted communication so contamination cannot move laterally.

- **Layer guardrails between intent and action:** Pre-execution input filters, runtime alignment controls, and post-execution output filters keep agent behavior aligned even when prompts or contexts are adversarial. Critical actions stay gated behind human approval.
- **Continuously test the security posture:** AI-aware vulnerability scanning, multi-agent red-team simulations, and runtime threat detection run as part of the lifecycle, not as one-time exercises.

Secure agent memory and state

One reason agentic AI is effective is the ability to perform long-running tasks. To achieve a goal, agents need to store context, tasks, and results. Agent memory systems can be targeted to introduce false information or influence agent decisions, so securing memory and state helps maintain agent integrity.

AGENTSEC01: How do you secure agentic memory and securely manage state between agents?

Capability intent

- Agent memory is partitioned along the isolation axes that matter for the workload, with each principal restricted to the namespaces its task requires.
- Tamper detection runs on every memory read, and an append-only versioned history supports forensic replay when a poisoning attempt is suspected.
- Every write path into memory, including user inputs, tool outputs, inter-agent messages, and consolidation, passes through a layered validation pipeline before data reaches the store.
- Hallucinations are detected and contained before they cascade across sessions or propagate to downstream agents in a multi-agent workflow.
- Memory operations are monitored continually, and anomalous access or injection signals are routed to the incident response pipeline for review.

Maturity levels

These levels summarize what each stage of maturity looks like for secure agent memory and state as a whole.

Level	Name	What it looks like
1	Initial	Memory is shared across agents, users, or sessions with no namespace partitioning, stored without encryption, and written without validation. Hallucinations propagate unchecked, cross-session contamination is indistinguishable from normal operation, and there is no tamper detection on reads.
2	Emerging	Memory is partitioned per session or per user with basic identity and access management (IAM) scoping on the store, and encryption at rest is turned on. Syntactic validation checks types, lengths, and formats before memory writes, and memory events flow to centralized logs. Detection is largely reactive and depends on human review.
3	Defined	Memory resources use hierarchical namespace schemas, for example Amazon Bedrock AgentCore Memory with per-actor and per-session placeholders, to partition data automatically. Multi-layer validation combines schema checks with

Level	Name	What it looks like
		Amazon Bedrock Guardrails for policy enforcement and personally identifiable information (PII) filtering, and contextual grounding checks run on high-stakes outputs before consolidation.
4	Proactive	Cryptographic integrity verification using HMAC signatures backed by AWS KMS runs on every memory read. All write paths share the same validation pipeline, including tool outputs and inter-agent messages. Anomaly detection in Amazon CloudWatch alarms on unusual access patterns, and a hallucination circuit breaker broadcasts through Amazon EventBridge to quarantine affected entries across downstream agents.

Level	Name	What it looks like
5	Optimized	Memory architecture is validated through routine red-team exercises that simulate poisoning and propagation. Validation policies and grounding thresholds adapt based on observed patterns, and detection signals feed directly into automated containment. Memory governance is codified and auditable.

Common issues to watch for

- Shared namespaces treated as the default rather than an explicit design decision, so one affected session can read or overwrite context that belongs to a different user or tenant.
- Validation applied only at the public API boundary, whereas tool outputs, inter-agent messages, and consolidation writes bypass the pipeline. This creates a blind spot for content that originates inside the agent system.
- Model outputs written directly into long-term memory without grounding checks, so hallucinated facts persist across sessions and compound because each agent builds its reasoning chain on fabricated context from earlier runs.
- Monitoring that captures memory events but routes no signals into alerts or incident response, so poisoning and propagation are discovered only during post-incident review.
- Memory access policies that are set once at deployment and never revisited, so permissions drift as the agent system evolves and the least-privilege posture decays over time.

Best practices

- [AGENTSEC01-BP01 Implement memory isolation and integrity controls](#)
- [AGENTSEC01-BP02 Validate and sanitize memory inputs](#)
- [AGENTSEC01-BP03 Monitor for hallucination propagation](#)

AGENTSEC01-BP01 Implement memory isolation and integrity controls

Shared agent memory is the shortest path for a single affected session to contaminate every other one. Partitioning memory along the boundaries that matter for the workload, and verifying what comes back out, contains the scope of memory poisoning to the partition it touched.

Desired outcome:

- You partition agent memory along the isolation axes that match the workload (session, user, tenant, agent, or group), with no cross-contamination between contexts.
- You detect unauthorized modifications to stored memories through integrity checks and keep a tamper-evident history of state changes.
- You scope memory access per agent through least-privilege IAM policies, so each agent can read or write only the namespaces it is authorized for.

Common anti-patterns:

- Sharing memory across agents or sessions by default, so an affected session can read or overwrite another's context and poisoning spreads laterally.
- Storing agent memory in plaintext without encryption at rest, exposing reasoning context, intermediate tool results, and user data if the underlying storage is reached through a misconfigured IAM policy or storage exposure.
- Skipping integrity checks on memory reads, letting silently corrupted or injected memories influence decisions and compound across sessions.
- Granting every agent broad read/write access to the full memory store, producing a flat trust model where any affected agent reaches unrelated workflows.

Benefits of establishing this best practice:

- Per-session and per-agent memory partitioning contains the scope of any single affected agent.
- Cryptographic integrity verification on memory reads catches poisoning attempts before affected data influences decisions.
- IAM-scoped memory access limits each agent to only the partitions it requires, reducing lateral movement.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Memory poisoning is a lateral-movement problem. If every agent reads from the same pool, one affected write becomes every agent's input, and the cost of a single successful injection is the correctness of the entire system. The design shift is to treat memory like a tenant-aware data store from the start. Partition it along the boundaries that actually matter for the workload:

- Session
- User
- Tenant
- Agent
- Group

Make cross-partition access the exception you must explicitly grant, not the default you must explicitly prevent.

[Amazon Bedrock AgentCore Memory](#) expresses this partitioning through a hierarchical namespace system built on actorId and sessionId identifiers, with dynamic placeholders like {actorId} and {sessionId} that resolve at call time. A namespace template like /support-agent/{actorId}/preferences gives each user an automatically scoped partition without hardcoding identifiers. The invoking session determines which namespace the agent reads from or writes to, and the IAM policy on the agent's role bounds which namespace prefixes it can touch, so even a namespace constructed incorrectly by the agent is denied by the authorization layer.

Shared namespaces are not an anti-pattern when they are intentional. A common product knowledge base read by every support agent, a team-scoped working context, or a coordination memory where cooperating agents exchange intermediate results are legitimate shared partitions. Model them as named namespaces (for example, /support-agent/shared/product-knowledge or /team-{teamId}/shared/working-context) and scope each agent role's IAM policy to the specific shared namespaces that agent is authorized to reach. The failure mode to help prevent is sharing by default, not sharing that has been designed and authorized.

Integrity is the other half of the pattern. AgentCore Memory keeps an append-only audit trail where the consolidation process marks outdated records as INVALID rather than deleting them, so the full sequence of changes is recoverable for forensic analysis. That helps protect history, but it doesn't detect whether a specific record was silently modified outside the normal consolidation flow. AWS KMS HMAC signatures layer tamper detection on individual records: sign on write,

recompute and compare on read, and treat a mismatch as evidence the record was altered. Pair that with customer-managed AWS KMS keys on the memory resource itself (through the `encryptionKeyArn` parameter) for sensitive reasoning context, and the built-in memory strategies filter personally identifiable information (PII) from long-term records by default.

Monitoring completes the control. Route memory access events to Amazon CloudWatch Logs, alarm on cross-namespace retrieval attempts and high-frequency writes from a single agent, and run red-team exercises that simulate memory poisoning to verify isolation controls hold under pressure.

Implementation steps

1. **Create the memory resource with customer-managed encryption:** Provision an [Amazon Bedrock AgentCore Memory](#) resource, specify `encryptionKeyArn` with a customer-managed AWS KMS key for sensitive workloads, and set `eventExpiryDuration` to match your data retention requirements.
2. **Design a hierarchical namespace schema:** Use dynamic placeholders (`{actorId}`, `{sessionId}`) to partition memory per user and per session automatically, for example `/support-agent/{actorId}/preferences` for user-scoped data and `/support-agent/shared/product-knowledge` for intentional shared context.
3. **Configure memory strategies at the correct isolation level:** Apply semantic, user preferences, summary, or custom strategies with namespaces scoped to the partition you want, and use custom strategy overrides where domain-specific extraction and consolidation prompts are needed.
4. **Scope IAM roles to authorized namespaces only:** Give each agent a dedicated IAM role with resource-based policies that allow only the namespace prefixes it requires, denying all other namespaces by default.
5. **Enforce namespace-scoped retrieval in agent code:** Call `retrieve_memory_records` and `list_events` within the agent's authorized namespaces only, so cross-tenant data is blocked at the application layer as well as the authorization layer.
6. **Layer HMAC integrity verification on sensitive records:** For workloads that need tamper detection beyond the built-in audit trail, store a KMS-generated HMAC alongside each memory entry and verify it on read before the content influences agent decisions.
7. **Alarm on memory access anomalies:** Configure Amazon CloudWatch alarms for cross-namespace retrieval attempts, high-frequency writes from a single agent, and spikes in memory extraction failures, and route alerts through Amazon EventBridge for automated incident response.

8. **Audit strategies and run red-team exercises:** Periodically review extraction and consolidation patterns with `list_memories` and `retrieve_memory_records` to verify strategies are capturing the right information and that PII filtering is working, and simulate memory poisoning scenarios to validate the isolation controls hold.

Resources

Related best practices:

- [AGENTSEC01-BP02 Validate and sanitize memory inputs](#)
- [AGENTSEC01-BP03 Monitor for hallucination propagation](#)
- [AGENTSEC03-BP03 Implement least privilege with dynamic boundaries](#)

Related documents:

- [Amazon Bedrock AgentCore Memory documentation](#)
- [Amazon Bedrock AgentCore Memory: Building context-aware agents](#)
- [Building smarter AI agents: AgentCore long-term memory deep dive](#)
- [AWS IAM best practices](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Key Management Service](#)
- [AWS Identity and Access Management](#)
- [Amazon DynamoDB](#)
- [Amazon CloudWatch](#)

AGENTSEC01-BP02 Validate and sanitize memory inputs

Unvalidated writes into agent memory let adversarial content persist and influence every subsequent session that reads the same context. Layered validation at ingestion keeps the memory store free of injection payloads, policy-violating content, and context inconsistent with the current task.

Desired outcome:

- You validate all data entering agent memory for type, format, and content before storage, and reject or quarantine policy-violating inputs before they influence agent behavior.
- You detect and block injection attempts at the memory ingestion layer and route suspicious inputs to human review.
- Your memory store contains only schema-conformant, sanitized data that downstream agents can consume safely.

Common anti-patterns:

- Storing raw, unvalidated user inputs directly into agent memory, letting prompt injection payloads persist and influence future sessions as agents build reasoning chains on top of affected context.
- Validating only at the public API boundary while skipping validation for content that enters memory from other write paths, including tool outputs, inter-agent messages, and memory consolidation.
- Failing to scan for encoded or obfuscated injection payloads, missing base64-encoded instructions, Unicode homoglyph substitutions, and other obfuscation that bypasses keyword-based filters but is still interpreted by downstream models.

Benefits of establishing this best practice:

- Multi-layer validation catches issues at syntactic, semantic, and contextual levels before data reaches the memory store.
- Blocking adversarial content at the ingestion boundary helps prevent it from influencing agent reasoning or propagating to downstream agents.
- Validation metrics surface trends in rejection rates and issue patterns, turning ingestion controls into operational signal.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Every write into agent memory is a trust decision. Raw user inputs are the obvious write path, but they are not the only one. Tool outputs arrive from search APIs, databases, and third-party services

the agent queries. Inter-agent messages carry content one agent wrote into another's scope, and memory consolidation generates long-term records by summarizing or merging existing events. Each of these is a distinct ingestion path, and each needs the same validation treatment. Validating only at the public API boundary leaves the other three open.

A layered pipeline gives each category of issue somewhere to be caught. Syntactic validation against a JSON Schema rejects wrong types, over-long strings, and missing fields before anything semantic happens. Semantic validation with [Amazon Bedrock Guardrails](#) detects prompt injection attempts, denied topics, and content that violates organizational guidelines through the ApplyGuardrail API, which evaluates content independently of model invocations so you can run it at any point in the pipeline. Contextual validation checks whether an input is consistent with the current task and flags anomalies for review.

[Amazon Bedrock AgentCore Memory](#) gives this pipeline a natural integration point. Because long-term memory extraction runs asynchronously from event ingestion, running Guardrails before events are written through the `create_event` API blocks harmful content from entering the extraction pipeline, and running Guardrails again before consolidation catches anything that made it past the first check. The built-in memory strategies already filter PII from long-term records by default, but that isn't a substitute for injection and policy enforcement, which must be added on top.

The shared responsibility model matters here. AWS is responsible for the AgentCore Memory infrastructure. You are responsible for secure application development, input validation, and helping prevent prompt injection in the memory extraction service. The [AgentCore Memory best practices](#) specifically recommend sanitizing user input with guardrails before persisting through `CreateEvent`. If your memory writes originate from HTTP APIs you already operate, AWS WAF in front of those APIs adds a network-layer tier, and Amazon API Gateway request validation enforces schema constraints at the same layer. For writes that happen entirely in agent code through direct SDK calls, validating in the agent code before `create_event` is the simpler path.

Failed inputs need a tiered response. Clearly harmful inputs are blocked and logged. Ambiguous inputs go to an Amazon SQS quarantine queue for human review, stored with enough context (agent ID, session ID, timestamp, and source) to support investigation. All validation failures emit Amazon CloudWatch metrics so rejection-rate trends become visible and configurable into alarms when something changes.

Implementation steps

1. **Define JSON schemas for every memory input type:** Specify field types, length limits, allowed values, and required fields so the first validation layer can reject malformed inputs deterministically.
2. **Configure Guardrails for semantic validation:** Configure [Amazon Bedrock Guardrails](#) with denied topics, word filters, and sensitive information filters tuned to your security policies, and use the ApplyGuardrail API so validation is independent of model invocations.
3. **Validate every write path, not just user input:** Apply Guardrails to tool outputs and inter-agent messages as well as user-provided content before any of them reach the memory store.
4. **Validate before create_event:** Run validation on events before they enter [Amazon Bedrock AgentCore Memory](#) short-term storage through create_event, so harmful content doesn't enter the asynchronous long-term extraction pipeline.
5. **Add AWS WAF on API-fronted memory writes:** Deploy AWS WAF managed rule groups on Amazon API Gateway endpoints that accept memory inputs, enforcing network-layer injection filtering before requests reach application code.
6. **Quarantine ambiguous inputs for review:** Route failures into an Amazon SQS queue with agent ID, session ID, timestamp, and source, so humans can review without blocking the pipeline.
7. **Emit validation telemetry:** Publish Amazon CloudWatch metrics for every validation outcome (pass, block, or quarantine) and alarm on elevated rejection rates that suggest an active issue.
8. **Review quarantined inputs regularly:** Use the quarantine queue to identify new attack patterns, update Guardrail configurations, and refine validation rules over time.
9. **Test for injection continually:** Apply penetration testing, static code analysis, and dynamic application security testing (DAST) to the memory write paths as part of regular security validation.
10. **Enforce IAM conditions on CreateEvent:** Use IAM Access Analyzer to validate that memory resource policies follow least privilege, and add policy conditions that restrict which roles can call the CreateEvent API for specific AgentCore Memory resources.

Resources

Related best practices:

- [AGENTSEC01-BP01 Implement memory isolation and integrity controls](#)
- [AGENTSEC01-BP03 Monitor for hallucination propagation](#)

- [AGENTSEC08-BP01 Multi-layer input validation and prompt injection defense](#)

Related documents:

- [Amazon Bedrock Guardrails documentation](#)
- [Amazon Bedrock AgentCore Memory best practices](#)
- [Amazon Bedrock AgentCore Memory: Building context-aware agents](#)
- [AWS WAF developer guide](#)

Related services:

- [Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore](#)
- [AWS WAF](#)
- [Amazon SQS](#)
- [Amazon CloudWatch](#)
- [Amazon API Gateway](#)

AGENTSEC01-BP03 Monitor for hallucination propagation

A single hallucinated fact stored as memory becomes ground truth for every agent that reads it next. Continuous grounding checks and confidence scoring keep fabricated content from entering memory or cascading across a multi-agent workflow.

Desired outcome:

- You detect false information before it propagates through agent memory or cascades across multi-agent workflows.
- You use confidence scoring to surface low-certainty outputs for validation, and fact-checking to help prevent hallucinated data from being stored as ground truth.
- When hallucination propagation is detected, affected memory entries are flagged or quarantined, and downstream agents are notified to discard potentially corrupted context.

Common anti-patterns:

- Storing model outputs directly into memory without a confidence threshold or grounding check, letting hallucinated facts persist and influence future decisions.
- Relying on the generating model to self-report uncertainty, which produces confident-sounding assessments even for fabricated content.
- Failing to propagate hallucination flags to downstream agents that consume shared memory, so corrupted context silently spreads through the workflow and each agent amplifies the error.
- Not logging hallucination detection events, reducing the risk of measurement of frequency or impact and blocking teams from tuning detection thresholds or identifying systemic patterns.

Benefits of establishing this best practice:

- Early detection catches hallucinated outputs before they propagate to downstream agents and compound into systemic errors.
- Confidence scoring gives a quantitative basis for deciding whether outputs are safe to store and act on.
- Ongoing monitoring surfaces new hallucination patterns for threshold tuning and rule refinement over time.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Hallucinations compound. A fabricated fact stored during one session is retrieved as context in the next, and the second agent, reasoning on that input, produces a second output that looks self-consistent with a false premise. In multi-agent systems the problem is worse because each downstream consumer treats shared memory as ground truth. The design response is to catch fabrications at the point they are about to enter memory, flag them with evidence, and propagate that flag to anything that already read the affected context.

[Amazon Bedrock Guardrails](#) contextual grounding is the first layer. It scores each model output against a provided reference context and rejects or flags anything below the threshold. Safety-critical applications run with higher thresholds, and creative tasks can run with lower ones. Pair contextual grounding with an LLM-as-a-Judge pattern for complex reasoning chains: route outputs through a secondary model invocation that receives the original context, the agent's output, and a structured evaluation prompt, and returns a confidence assessment. Keyword matching alone isn't

sufficient at this layer. The judge catches contradictions and unsupported claims that simple filters miss.

[Amazon Bedrock AgentCore Memory](#) gives the check a natural home. The long-term memory consolidation process retrieves semantically similar existing memories and uses an LLM to decide whether to add, update, or skip new information, and outdated memories are marked as INVALID rather than deleted. That produces an immutable trail you can walk to trace how hallucinated content entered and propagated. Running grounding checks before `create_event` helps keep fabrications out of the extraction pipeline, and custom memory strategy overrides let you bake grounding validation into the extraction and consolidation prompts for your domain.

Detection without traceability is expensive. Enable Amazon Bedrock model invocation logging and build Amazon CloudWatch Logs Insights queries that look for hallucination indicators (references to non-existent resources, contradictory statements within a single response, outputs that deviate significantly from input context). [Amazon Bedrock AgentCore Observability](#) provides a session, trace, and span hierarchy that lets you correlate a session-level anomaly back to the specific span where the hallucinated content originated. AgentCore emits default span data for memory resources, viewable in Amazon CloudWatch Logs and Amazon CloudWatch Application Signals, and session-level metrics are available on the CloudWatch generative AI observability page. For deeper visibility, instrument agent code with AWS Distro for OpenTelemetry (ADOT) to capture custom metrics for grounding scores, confidence thresholds, and validation outcomes at each step.

The circuit breaker keeps a single hallucination from cascading. When detection fires in one agent, flag every memory entry that agent wrote during the current session for re-validation before downstream agents consume it, and broadcast the detection event through Amazon EventBridge so every agent in the workflow can discard potentially corrupted context. Tag memory entries with confidence scores and grounding results so the evidence basis for every decision is auditable.

Implementation steps

1. **Configure contextual grounding thresholds per use case:** Set [Amazon Bedrock Guardrails](#) contextual grounding thresholds that match each agent's risk profile, with higher thresholds for safety-critical applications and lower ones for creative tasks.
2. **Add an LLM-as-a-Judge step for high-stakes outputs:** Route outputs through a secondary model invocation that evaluates factual consistency against the original context before the output is committed to memory.

3. **Run grounding checks before create_event:** Apply grounding validation at the [Amazon Bedrock AgentCore Memory](#) event ingestion boundary, so hallucinated content is filtered before reaching the long-term extraction and consolidation pipeline.
4. **Use custom memory strategy overrides for domain-specific grounding:** Incorporate grounding validation logic into the extraction and consolidation prompts through custom strategy overrides where your domain has specific factuality requirements.
5. **Enable Amazon Bedrock model invocation logging:** Turn on Amazon Bedrock model invocation logging and create Amazon CloudWatch Logs Insights queries that detect references to non-existent resources, contradictory statements, and significant deviations from input context.
6. **Alarm on output-consistency anomalies:** Configure Amazon CloudWatch anomaly detection on output-consistency metrics to baseline normal patterns and alert on deviations that suggest systematic hallucination.
7. **Instrument with ADOT and AgentCore Observability:** Use AWS Distro for OpenTelemetry to capture custom spans for grounding scores and validation outcomes, and use the session/trace/span hierarchy in [Amazon Bedrock AgentCore Observability](#) to correlate detections back to the originating interaction.
8. **Wire a circuit breaker for propagation:** When a hallucination fires, flag every memory entry from the current session and broadcast the detection event through Amazon EventBridge so downstream agents can discard potentially corrupted context.
9. **Tag memory entries with evidence:** Store confidence scores and grounding check results alongside each memory entry to produce an auditable record of the evidence basis for agent decisions.
10. **Review detection logs periodically:** Tune thresholds, update detection rules, and identify systemic patterns by reviewing hallucination detection logs on a regular cadence.

Resources

Related best practices:

- [AGENTSEC01-BP01 Implement memory isolation and integrity controls](#)
- [AGENTSEC01-BP02 Validate and sanitize memory inputs](#)
- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)
- [AGENTREL05-BP03 Ground agent cognition in real information](#)

Related documents:

- [Amazon Bedrock Guardrails contextual grounding](#)
- [Amazon Bedrock model invocation logging](#)
- [AgentCore Observability: Sessions, traces, and spans](#)
- [Amazon Bedrock AgentCore Memory best practices](#)
- [Building smarter AI agents: AgentCore long-term memory deep dive](#)

Related services:

- [Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [Amazon EventBridge](#)

Secure agent tool usage

Agents interact with external systems through tools such as APIs, databases, file systems, and other services. Without proper controls, an agent with tool access can be directed to perform unintended actions. Authorization, input validation, and tool governance reduce the risk of tool misuse and unauthorized operations.

AGENTSEC02: How do you control and secure agent tool usage?

Capability intent

- Every tool invocation is authorized against a declarative policy before execution, with the agent identity and the originating user context propagated through the authorization chain.
- Agents invoke only tools within their approved scope, and tool parameters and responses pass through schema and policy checks that keep operations inside defined boundaries.
- High-risk mutating operations stop at a human-in-the-loop checkpoint, and rate limits cap the impact scope when an agent enters a runaway loop.

- Tools and the MCP servers that host them are available to agents only after going through a documented security review and being registered with a pinned version, data classification, and review expiry.
- Tool usage is observable end to end, with authorization decisions, validation failures, and invocation rates feeding detection and compliance workflows.

Maturity levels

These levels summarize what each stage of maturity looks like for secure agent tool usage as a whole.

Level	Name	What it looks like
1	Initial	Agents are granted blanket access to available tools, and authorization is delegated to the agent's own reasoning . Parameters flow to tools without schema enforcement, error responses return raw stack traces, internal endpoint addresses, or database schema details to the agent's context, and there is no registry or version control over tools or the MCP servers that host them. High-risk writes, deletes, and financial actions execute without review.
2	Emerging	Each agent runs under a dedicated identity with scoped permissions on the tools it needs. Schema validation is applied in the tool handler for user-

Level	Name	What it looks like
		facing parameters, error responses are sanitized so internal details don't reach the agent or the user, and AWS Lambda timeouts and memory allocations bound the runtime cost and execution time of any single tool invocation. Tools and the MCP servers hosting them are documented in a shared list, but enforcement depends on reviewer discipline.
3	Defined	Agent-to-tool traffic flows through Amazon Bedrock AgentCore Gateway , which centralizes OAuth-based inbound authorization and target-specific outbound authentication. Tools and the MCP servers that host them are registered in a version-controlled registry with documented permissions, data classification, and review expiry, and strict tool use constrains model-generated parameters to conform to the tool schema at the decoding layer.

Level	Name	What it looks like
4	Proactive	Policy in Amazon Bedrock AgentCore enforces Cedar policies with a default-deny posture at the gateway, evaluating every tool call against identity-aware conditions and parameter constraints outside the agent's reasoning loop. Human-in-the-loop checkpoints intercept high-risk mutating operations through AWS Step Functions callback patterns, rate limits run at both the gateway and tool tiers, and Amazon CloudWatch alarms fire on authorization failures and validation spikes.

Level	Name	What it looks like
5	Optimized	Tool authorization, validation, and registry controls are codified end to end and validated continuously through red-team exercises and automated drift detection with AWS Config . Deprecation workflows remove expired tools and MCP server entries automatically, policies adapt from observed behavior, and centralized MCP server governance (covering registration, version pinning, network egress controls, and shared discovery) is enforced consistently across accounts.

Common issues to watch for

- Tool access granted at the agent level but enforced only by the agent's reasoning, so nothing outside the model helps prevent an injected or poisoned prompt from invoking tools that sit outside the agent's intended scope.
- Schema validation applied to user-facing parameters but not to parameters the model generates, on the assumption that the model can't produce malformed output. In practice, model-generated parameters are an attack surface that carries injection payloads and invalid types.
- Registries maintained as static documents rather than as enforced controls at the gateway or the MCP client, so agents continue to discover and invoke tools that no one has reviewed.
- Remote Model Context Protocol (MCP) servers adopted with the same risk assumptions as local tools, which understates the exposure introduced by network connectivity, third-party authentication, and external data handling.

- Rate limiting treated as a cost control rather than as a security control, so a runaway loop or a prompt injection that triggers mass tool calls runs long enough to affect downstream systems before human responders intervene.

Best practices

- [AGENTSEC02-BP01 Implement tool authorization](#)
- [AGENTSEC02-BP02 Validate tool inputs and outputs](#)
- [AGENTSEC02-BP03 Maintain approved tool registry with security assessments](#)

AGENTSEC02-BP01 Implement tool authorization

An agent with unconstrained tool access has no meaningful privilege boundary. Externally enforced authorization at the gateway, combined with identity propagation and human review for mutating operations, enforces bounded autonomy at the tool layer.

Desired outcome:

- You authorize every tool invocation against a defined policy before execution, with agent identity and user context propagated through the authorization chain.
- Agents can invoke only the tools within their approved scope, and attempts to access unauthorized tools are blocked and logged.
- Human-in-the-loop checkpoints intercept high-risk mutating operations so consequential actions receive review before execution.

Common anti-patterns:

- Granting agents blanket access to every available tool rather than scoping access to the tools each agent requires.
- Relying on the agent's own judgment to decide whether a tool invocation is appropriate, with no independent check at the tool or API layer.
- Failing to propagate user identity context through tool invocations, so downstream services can't enforce user-level access controls and every call runs with the agent's permissions.
- Skipping human-in-the-loop controls for mutating operations because they add latency, accepting unbounded risk for actions that are difficult or impossible to reverse.

Benefits of establishing this best practice:

- RBAC policies scope each agent to the tools its defined tasks require, implementing least-privilege tool access.
- Identity propagation lets downstream services enforce user-level access controls on resources reached through agent tool calls.
- Rate limiting and human-in-the-loop controls constrain autonomous execution to operations with acceptable risk profiles.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Authorization implemented only through prompt instructions is insufficient because prompts can be manipulated through adversarial phrasing or prompt injection. Implement authorization as an external, deterministic check that happens before the tool executes, regardless of how the agent arrived at the call. [Amazon Bedrock AgentCore Gateway](#) is the enforcement point, and [Policy in Amazon Bedrock AgentCore](#) is the rules engine.

Gateway runs a dual-sided security model. On the inbound side, it follows the MCP authorization specification and acts as an OAuth resource server, working with Amazon Cognito, Okta, Auth0, or your own OAuth provider. You configure approved client IDs and audiences to control which applications and agents can reach your tools, and Gateway supports both authorization code flow (3LO) and client credentials flow (2LO) for service-to-service communication. On the outbound side, the authentication model depends on target type: AWS Lambda and Smithy model targets use IAM-based authorization through a role you configure with scoped permissions, and OpenAPI targets support API key or OAuth 2LO client credentials grant. [Amazon Bedrock AgentCore Identity](#) resource credentials providers handle token caching and secure storage, and each target is associated with exactly one authentication configuration for clear boundaries and auditability.

Policy is where fine-grained authorization lives. Cedar policies evaluate every agent-to-tool request at the gateway before execution, with a default-deny posture where forbid always wins over permit. Conditions can reference OAuth claims from the JWT token (user role, scopes, tenant-level identifiers like patient ID), tool input parameters, and runtime context such as time of day. That lets you express rules like "role=clinician can reschedule appointments for patients in their own panel" as a deterministic policy rather than a hope about the prompt. Policies can be authored directly in Cedar or generated from natural language, and Gateway supports a LOG_ONLY mode so you can validate policy behavior against live traffic before switching to enforce mode.

Tool overload is its own risk. Presenting an agent with hundreds of tools increases the chance it selects the wrong one or follows an inefficient execution path. Gateway's built-in `x_amz_bedrock_agentcore_search` tool exposes semantic tool discovery so agents locate relevant tools through natural language rather than seeing the full inventory. That reduces the surface the model reasons across on any given turn.

For tools that perform mutating operations (writes to databases, outbound emails, financial transactions), human-in-the-loop review belongs in the execution path, not the prompt. AWS Step Functions callback patterns let an agent pause and wait for approval. The workflow sends an approval request through Amazon SNS or Amazon SES and resumes only after a human responds within a defined timeout. Configure escalation paths for timeouts so a non-response doesn't silently block a legitimate action. Rate limiting at both the Gateway and tool API levels helps prevent resource exhaustion: Amazon API Gateway usage plans and throttling enforce per-agent rate limits, and AWS Lambda reserved concurrency caps the maximum parallel tool invocations. Gateway publishes usage, invocation, performance, and error metrics to Amazon CloudWatch and integrates with AWS CloudTrail for a full audit trail, so runaway loops and unexpected call patterns surface as signal.

Implementation steps

1. **Configure Gateway inbound OAuth:** Create an [Amazon Bedrock AgentCore Gateway](#) and wire inbound OAuth authorization to your identity provider (Amazon Cognito, Okta, Auth0, or your own OAuth provider), specifying approved client IDs and audiences.
2. **Add targets with scoped outbound credentials:** Register tool APIs as gateway targets, configuring IAM roles for Lambda and Smithy targets and API key or OAuth 2LO for OpenAPI targets, and manage outbound credentials through [Amazon Bedrock AgentCore Identity](#) resource credentials providers.
3. **Author Cedar policies for authorization:** Create a policy engine in [Policy in Amazon Bedrock AgentCore](#) and define Cedar policies with identity-aware conditions on OAuth claims (user role, scopes, user ID) and tool input parameters. Author directly in Cedar or generate policies from natural language descriptions.
4. **Validate policies in LOG_ONLY before enforcing:** Associate the policy engine with Gateway in LOG_ONLY mode, review observability logs to confirm the policies produce the expected permit and deny decisions, then switch to enforce mode.
5. **Enable semantic tool discovery:** Opt in to the built-in `x_amz_bedrock_agentcore_search` tool so agents locate relevant tools through natural language queries rather than reasoning over the full inventory.

6. **Add human-in-the-loop approvals for mutating tools:** Wire AWS Step Functions callback patterns for high-risk tools, send approval requests through Amazon SNS or Amazon SES, and configure escalation paths for reviewer timeouts.
7. **Cap concurrency and request rates:** Enforce per-agent rate limits through Amazon API Gateway usage plans and cap parallel invocations with AWS Lambda reserved concurrency to help prevent resource exhaustion.
8. **Monitor authorization decisions:** Use Gateway's Amazon CloudWatch metrics and AWS CloudTrail integration to track tool invocations, authorization failures, and rate-limit events, and configure alarms for authorization-failure spikes.
9. **Review tool authorization quarterly:** Remove unused permissions and tighten access boundaries on a regular cadence as workloads and tools evolve.

Resources

Related best practices:

- [AGENTSEC02-BP02 Validate tool inputs and outputs](#)
- [AGENTSEC02-BP03 Maintain approved tool registry with security assessments](#)
- [AGENTSEC03-BP03 Implement least privilege with dynamic boundaries](#)
- [AGENTREL02-BP02 Limit agent permissions to minimum required access](#)
- [AGENTCOST04-BP01 Design cost effective tool selection to minimize unnecessary invocations](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway documentation](#)
- [Introducing Amazon Bedrock AgentCore Gateway: Transforming enterprise AI agent tool development](#)
- [Apply fine-grained access control with Bedrock AgentCore Gateway interceptors](#)
- [Secure AI agents with Policy in Amazon Bedrock AgentCore](#)
- [Amazon Bedrock AgentCore Identity documentation](#)

Related examples:

- [Healthcare appointment agent with Policy enforcement \(GitHub\)](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Step Functions](#)
- [Amazon API Gateway](#)
- [Amazon CloudWatch](#)
- [AWS CloudTrail](#)

AGENTSEC02-BP02 Validate tool inputs and outputs

Agents generate tool parameters from model output, which means malformed or adversarial inputs can reach tools through ordinary reasoning, not just through unauthorized callers. Schema-driven validation on inputs and sanitization on outputs keep tools operating inside their intended parameter space and help prevent error messages from disclosing internal system details.

Desired outcome:

- You validate every parameter passed to tools against a defined schema before execution, and sanitize tool outputs before returning them to the agent.
- Injection through tool parameters is blocked, oversized inputs are prevented from exhausting resources, and error messages are sanitized to avoid leaking sensitive system information.
- Tool invocations operate predictably within defined boundaries, and validation failures are logged for security analysis.

Common anti-patterns:

- Passing raw agent-generated parameters directly to tools without type checking or range validation, letting malformed inputs cause unexpected behavior or injection.
- Returning raw tool error messages to the agent without sanitization, exposing internal system details, stack traces, or infrastructure information usable for further probing.
- Validating only user-provided inputs and skipping validation for parameters produced by the agent's reasoning, on the assumption the model can't generate malformed output.

Benefits of establishing this best practice:

- Schema-enforced input validation helps prevent tools from operating outside their intended parameter space.
- Sanitized error responses return failure categories without exposing internal system details.
- Timeout controls, memory limits, and output-size enforcement help prevent resource exhaustion from oversized or long-running tool invocations.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Validation is more useful when it happens at several layers than when any single layer tries to do all the work. The cheapest and most specific layer is at the model itself. Amazon Bedrock structured outputs with strict tool use constrain the model's decoding so that generated tool parameters always conform to the defined input schema. Setting `strict: true` on the tool definition, together with `additionalProperties: false` and enum constraints on fields with a closed set of values, helps prevent malformed parameters as a class before they ever reach the tool. That doesn't replace application validation, but it removes a large chunk of the work from it. The [Structured outputs on Amazon Bedrock blog](#) covers the parameter shapes and the model-level enforcement.

The next layer is schema validation in the tool invocation pipeline. For tools deployed as AWS Lambda functions, a JSON Schema check inside the Lambda handler, or a shared Lambda Layer, enforces type constraints, value ranges, string length limits, and format patterns before the function logic runs. This is the place to catch the edge cases strict tool use doesn't cover, anything involving relationships between fields, external-state constraints, or values the model can't know.

Policy in Amazon Bedrock AgentCore provides a third layer at the gateway. Cedar policies can evaluate conditions on tool input parameters through `context.input`, so business rules like "financial amount below an approved threshold" or "date parameter within an acceptable range" are enforced deterministically at the gateway before the call reaches the backend. The value of this layer is that the rules are auditable and managed independently of tool code. The value of keeping it separate from the first two layers is that changes to business rules don't require redeploying tools.

Logical constraints that are not expressible as schema or simple comparisons need a different mechanism. [Amazon Bedrock Guardrails](#) Automated Reasoning checks verify that tool parameters conform to logical constraints, a date range with start before end, a set of values that must be mutually consistent. Apply Guardrails at the AWS Organization level where consistent policy enforcement is required across all agent deployments.

Resource protection and error sanitization complete the picture. AWS Lambda function timeout and memory limits, sized to each tool's measured execution profile, bound what any single call can consume, and Lambda reserved concurrency caps total parallel invocations. Tool outputs that return large datasets need size limits and pagination. Truncation events belong in the log so an agent doesn't silently reason on a partial response. Error handling catches exceptions and returns structured responses that describe the failure category without exposing internal details, stack traces, or infrastructure information. AWS WAF managed rule groups on API-based tool endpoints add a network-layer filter for common injection patterns before requests reach tool code.

Implementation steps

1. **Constrain parameters at the model layer:** Enable strict tool use (`strict: true`) on tool definitions in Amazon Bedrock, set `additionalProperties: false` on all input schemas, and define enum constraints for fields with a limited set of valid values to block malformed parameters at decoding.
2. **Enforce schemas in the invocation pipeline:** Define JSON Schema specifications for every tool and validate parameters as a middleware layer inside the Lambda handler or a shared Lambda Layer before the tool function runs.
3. **Add Cedar policy checks at the gateway:** Define policies in AgentCore Policy with conditions on `context.input` to enforce business rules and parameter constraints deterministically, complementing application-level schema validation.
4. **Use Automated Reasoning for logical constraints:** Configure [Amazon Bedrock Guardrails](#) Automated Reasoning policies for tool inputs and outputs that require logical constraint validation beyond schema and Cedar rules.
5. **Right-size Lambda limits per tool:** Set AWS Lambda timeout and memory limits based on measured execution profiles, with conservative limits that help prevent resource exhaustion, and use reserved concurrency to cap parallel invocations.
6. **Sanitize errors:** Implement structured error handling in every tool that returns sanitized responses without internal system details, stack traces, or infrastructure information.
7. **Paginate and truncate large outputs:** Apply output size limits for tools that may return large datasets, truncate responses before returning them to the agent, and log every truncation event.
8. **Add AWS WAF in front of API-based tools:** Deploy AWS WAF with managed rule groups on API-based tool endpoints to filter common injection patterns at the network layer.
9. **Alarm on validation-failure rates:** Publish Amazon CloudWatch metrics for validation outcomes and configure alarms for elevated failure rates that suggest active injection attempts or misconfigured parameters.

Resources

Related best practices:

- [AGENTSEC02-BP01 Implement tool authorization](#)
- [AGENTSEC02-BP03 Maintain approved tool registry with security assessments](#)
- [AGENTSEC08-BP01 Multi-layer input validation and prompt injection defense](#)

Related documents:

- [Amazon Bedrock Guardrails automated reasoning checks](#)
- [Structured outputs on Amazon Bedrock: Schema-compliant AI responses](#)
- [Secure AI agents with Policy in Amazon Bedrock AgentCore](#)
- [AWS Lambda best practices](#)

Related services:

- [Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore](#)
- [AWS Lambda](#)
- [AWS WAF](#)
- [Amazon CloudWatch](#)
- [Amazon API Gateway](#)

AGENTSEC02-BP03 Maintain approved tool registry with security assessments

Every tool an agent can reach is part of its effective privilege surface. A version-controlled registry with documented security boundaries, enforced at invocation time, keeps unvetted or deprecated tools off the agent's call path.

Desired outcome:

- You maintain a centralized, version-controlled registry of approved tools, each with documented security boundaries, required permissions, data classification levels, and a current vulnerability assessment.
- Agents can access only tools present in the registry, and unapproved tools are blocked by default.
- You continually validate the registry for compliance, and deprecated tools are removed from agent access automatically.

Common anti-patterns:

- Allowing agents to discover and invoke any available tool or MCP server without prior security review and registry approval.
- Maintaining the tool registry as a static document rather than an enforced control, so agents can bypass it and invoke unapproved tools directly.
- Failing to distinguish between locally hosted tools and remote MCP servers in the risk assessment, underestimating the expanded scope from external network connectivity.
- Skipping version pinning for approved tools, so agents pick up new versions that have not undergone security review.

Benefits of establishing this best practice:

- A deny-by-default posture constrains agent capabilities to a pre-approved, security-reviewed set of tools and operations.
- Version control helps prevent agents from using tool versions that have not been reviewed and provides an audit trail of which versions were approved and when.
- Automated compliance checks detect drift from the approved registry and trigger remediation workflows.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

A tool registry must be enforced at runtime to be effective. Document approved tools in a registry and configure the invocation path to refuse tools that are not on the list. The design pattern is a registry that agents can't bypass: tools reachable only through a gateway target, agents authorized

only through a policy engine with default-deny semantics, and an out-of-band compliance process that detects drift between the registry and what is actually configured.

[Amazon Bedrock AgentCore Gateway](#) provides the consolidation point for agent-to-tool traffic. Each gateway target represents a backend service or group of APIs exposed as tools to agents, with defined tool schemas, authentication configurations, and access controls. Gateway alone isn't deny-by-default, however: adding a target makes it immediately accessible as an MCP tool to any agent that reaches the gateway endpoint. To restrict which agents can invoke which tools, layer [Policy in Amazon Bedrock AgentCore](#) Cedar policies with a default-deny posture, Gateway Interceptor for custom Lambda-based access logic, or both. The policy layer is what turns a populated registry into enforced authorization.

Development environments need a second control. When developers and agents interact with MCP servers through IDE-based tools, an MCP registry, a JSON allowlist of approved servers hosted on an HTTPS endpoint such as Amazon S3 or an internal web server, gives clients a list of servers to fetch at startup and re-sync periodically (typically every 24 hours). Servers not in the registry are blocked, and if a locally installed server is removed from the registry, the client terminates it and helps prevent it from being re-added. The registry supports version pinning so that a new version automatically relaunches clients with the updated version, and the file format follows the [MCP Registry open standard](#) so the investment isn't tied to a single tool or provider. MCP registry governance can be configured at the organization level with account-level overrides, for example disabling MCP for the organization by default but enabling it with a specific allowlist for certain teams.

At enterprise scale, a centralized MCP server hub consolidates what would otherwise become a proliferation of team-specific connections. Teams develop MCP servers for their specific functions, but servers are hosted centrally and accessible across the organization through a shared registry or discovery API backed by Amazon DynamoDB that catalogs available servers with their descriptions, tool definitions, and access requirements. Network-level access uses AWS PrivateLink and VPC endpoints so agents connect only to trusted organization-hosted servers, and each server runs as an isolated container on Amazon ECS with AWS Fargate for independent scaling without impact on other servers.

Remote MCP servers need a heightened security review. They introduce network connectivity to external services, expanding scope beyond the organization's direct control. Assess authentication mechanisms, data handling practices, and network exposure, and apply network controls such as VPC endpoints and security groups to restrict connectivity to the required endpoints. When onboarding tools to Gateway or the MCP registry, scan API specifications for security risks, validate

authentication, assess data handling, enrich tool metadata with descriptions, usage examples, and performance characteristics, and group APIs into gateway targets by business domain, outbound authorization requirements, and API type.

Gateway supports six target types:

- Lambda functions
- API Gateway REST APIs
- OpenAPI schemas
- Smithy models
- External MCP servers
- Built-in templates from integration providers

Built-in templates provide pre-configured, curated integrations for popular SaaS platforms including Salesforce, Slack, Jira, Asana, Zendesk, and ServiceNow, with a vetted subset of provider APIs exposed through the gateway. Routing all tool access through Gateway (internal services, external MCP servers, and native SaaS integrations) consolidates authentication, schema enforcement, and policy evaluation under one endpoint. IDEs such as Kiro, Claude Code, and Cursor connect through the Amazon Bedrock AgentCore MCP Server, which bridges IDE-based MCP clients to the gateway endpoint.

Continuous compliance detection keeps the registry enforceable over time. Maintain a configuration store in Parameter Store, a capability of AWS Systems Manager or AWS AppConfig alongside the gateway configuration, with entries for tool name, approved version, required IAM permissions, data classification level, security review date, and expiration date. Use AWS Config rules to validate that agent deployments reference only registry-approved tools, and trigger Amazon EventBridge notifications for non-compliance. Automated deprecation workflows remove expired tools from the registry, update agent configurations, and help prevent continued use.

Implementation steps

1. **Build the structured registry:** Create a tool registry in Parameter Store, a capability of AWS Systems Manager or AWS AppConfig with entries for each approved tool covering version, permissions, data classification, and review metadata.
2. **Add approved tools as Gateway targets:** For agent-to-tool traffic, register approved tools as [Amazon Bedrock AgentCore Gateway](#) targets with defined tool schemas, outbound

authentication configurations, and access controls. Group targets by business domain, authorization requirements, and API type.

3. **Publish an MCP registry for development:** Create an MCP registry JSON file that follows the [MCP Registry open standard](#), host it on an HTTPS endpoint, and configure it in your organization's admin settings with version pinning for each server entry.
4. **Define the security review process:** Establish a review covering API specification scanning, permission assessment, data flow mapping, and authentication mechanism validation, with findings documented in the registry entry and a review expiration date.
5. **Build a centralized hub at enterprise scale:** For multi-LOB deployments, implement a centralized MCP server hub with an Amazon DynamoDB-backed discovery API, network-level access through AWS PrivateLink and VPC endpoints, and isolated container hosting on Amazon ECS with AWS Fargate.
6. **Enforce default-deny through Policy:** Configure Cedar policies in AgentCore Policy so only explicitly permitted tools can be invoked, providing a second enforcement layer beyond Gateway target configuration.
7. **Apply heightened review to remote MCP servers:** Assess network exposure and external authentication, and apply VPC endpoints and security groups to restrict connectivity.
8. **Detect registry drift continually:** Deploy AWS Config rules to detect agent configurations referencing unapproved tools, and trigger Amazon EventBridge notifications for remediation.
9. **Automate deprecation:** Expire tools past their review date, remove them from Gateway targets and the MCP registry, and update agent configurations to help prevent continued use.

Resources

Related best practices:

- [AGENTSEC02-BP01 Implement tool authorization](#)
- [AGENTSEC02-BP02 Validate tool inputs and outputs](#)
- [AGENTSEC03-BP03 Implement least privilege with dynamic boundaries](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway documentation](#)
- [Introducing Amazon Bedrock AgentCore Gateway: Transforming enterprise AI agent tool development](#)

- [Transform your MCP architecture: Unite MCP servers through AgentCore Gateway](#)
- [Enterprise governance: control your MCP servers and models](#)
- [MCP governance for Q Developer](#)
- [Accelerating AI innovation: Scale MCP servers for enterprise workloads with Amazon Bedrock](#)
- [Secure AI agents with Policy in Amazon Bedrock AgentCore](#)
- [Parameter Store, a capability of AWS Systems Manager](#)

Related examples:

- [Accelerating AI Innovation: Scaling Model Context Protocol Servers for Enterprise Workloads on AWS \(GitHub\)](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Systems Manager](#)
- [AWS Config](#)
- [Amazon EventBridge](#)
- [AWS AppConfig](#)
- [Amazon ECS](#)
- [Amazon DynamoDB](#)

Agent identity and permission management

Agents access resources, invoke tools, and interact with other services in two patterns:

- Explicitly on behalf of a user, where a human initiated the request and the agent's actions must be bounded by that user's permissions
- Autonomously, where the agent acts without a user in the loop (triggered by a schedule, an event, an alarm, or another agent)

Identity and permission management needs to handle both. Without it, agent permissions can lead to unauthorized access to resources and data. To maintain security boundaries, apply least-

privilege principles, separate agent and human permissions, and set up strong authentication for agent identities.

AGENTSEC03: How do you manage agent identities, permissions, and prevent privilege escalation?

Capability intent

- Every agent-to-agent and agent-to-service communication authenticates through verifiable mechanisms, whether that is certificate-based mutual TLS, signed OAuth tokens, or platform-managed workload identity.
- Agents operate under service identities that are distinct from human identities, and audit trails attribute every action unambiguously to either an agent or a human actor.
- When an agent acts on behalf of a user, the user context propagates as signed token claims through the call chain without the agent ever assuming the user's credentials.
- Agents run with the minimum permissions their task requires, through short-lived credentials, permission boundaries, and IAM Conditions.
- Permission posture is continually validated, with automated drift detection, unused-access findings, and documented periodic reviews keeping privilege aligned with actual usage.

Maturity levels

These levels summarize what each stage of maturity looks like for agent identity and permission management as a whole.

Level	Name	What it looks like
1	Initial	Agents authenticate with shared API keys or static tokens, and some roles are reused across agents and human users. Permissions are broad, credentials are long-lived, and audit trails don't clearly distinguish

Level	Name	What it looks like
		agent actions from human actions. There is no permission boundary, no revocation path, and no review cadence.
2	Emerging	Each agent has a dedicated AWS Identity and Access Management (IAM) role with a consistent naming and tagging convention. Certificates from AWS Private Certificate Authority are used where mutual TLS is required, credentials rotate through AWS Secrets Manager , and AWS CloudTrail logs agent activity separately from human activity. Initial least-privilege policies are in place, although drift is common.

Level	Name	What it looks like
3	Defined	Amazon Bedrock AgentCore Identity centralizes agent workload identities, token issuance, and the token vault, with customer-managed AWS KMS keys protecting secrets. Human operators access AWS through AWS IAM Identity Center , and Service Control Policies (SCPs) in AWS Organizations help prevent agents from assuming human roles. AWS STS AssumeRole issues short-lived credentials with session policies, and IAM Access Analyzer provides baseline findings.

Level	Name	What it looks like
4	Proactive	When an agent acts on behalf of a user, GetWorkloadAccessTokenForJWT embeds user context as claims so that downstream services enforce user-level authorization without the agent holding the user's credentials. IAM permission boundaries cap every agent role, IAM Conditions restrict access by region, tag, time window, and source VPC, and just-in-time elevation with automatic revocation handles high-privilege operations. AWS Config and Amazon EventBridge alert on policy changes in near real time.

Level	Name	What it looks like
5	Optimized	Identity and permission governance is fully codified, with continuously validated least-privilege baselines derived from CloudTrail data and aggregated findings in AWS Security Hub CSPM. Unused-access and unusual-authentication findings feed automated remediation. Access reviews run on a cadence that matches the customer's risk profile, with timestamped, documented sign-off, and identity controls are validated through red-team exercises against impersonation and privilege-escalation paths.

Common issues to watch for

- Shared API keys or static tokens left in use for agent authentication, with no rotation cadence or revocation path defined. The credential becomes a long-lived secret distributed across environments, and any disclosure exposes every agent that holds it until each one is rotated.
- Agent roles and human roles blurred at the edges, whether through role reuse, role chaining, or missing trust-policy and SCP guardrails, so audit trails can't cleanly distinguish agent actions from human actions during incident investigation.
- Agent permissions expanded reactively in response to access-denied errors without investigating whether the access pattern is consistent with the agent's intended scope, leading to steady privilege creep that no single review catches and that pushes agent roles toward the broad permission posture autonomous agents should not have.
- Access reviews for agent identities conducted on a cadence inherited from human-user reviews (annual or post-incident), even though agent permissions can drift much faster as new tools,

prompts, and orchestration patterns are added, so reviewers can't distinguish permissions that are genuinely unused from permissions held for upcoming agent capability changes.

- Delegated access implemented by having the agent assume a user's role rather than carrying user context as signed token claims, which collapses the audit trail and gives the agent the user's full permission set for the duration of the session.

Best practices

- [AGENTSEC03-BP01 Implement strong authentication for agent identities](#)
- [AGENTSEC03-BP02 Separate agent and human user permission](#)
- [AGENTSEC03-BP03 Implement least privilege with dynamic boundaries](#)
- [AGENTSEC03-BP04 Regular permission audits and access reviews](#)

AGENTSEC03-BP01 Implement strong authentication for agent identities

Shared API keys and one-way TLS give agents enough network reachability to be useful and enough ambiguity to be impersonated. Cryptographic identity for both sides of every call, with automated lifecycle and immediate revocation, is the control that makes every agent communication auditable and reversible.

Desired outcome:

- You authenticate all agent-to-agent and agent-to-service communications using strong cryptographic mechanisms, with mutual authentication that helps prevent impersonation and interception.
- You automate certificate lifecycle management so expired certificates don't cause authentication failures or security gaps.
- You can revoke affected agent identities immediately, cutting off unauthorized access within minutes.

Common anti-patterns:

- Using shared API keys or static tokens for agent authentication instead of certificate-based or OAuth mechanisms, producing credentials that are hard to rotate and straightforward to exfiltrate.

- Implementing one-way TLS (server authentication only) without mutual authentication, so any client on a permitted network path can reach the endpoint without proof of its agent identity.
- Managing certificate lifecycles manually, leading to expired certificates that either cause outages or are renewed without proper security review.

Benefits of establishing this best practice:

- Certificate-based or OAuth authentication provides cryptographic proof of identity for both parties in every agent communication.
- Automated certificate lifecycle management reduces the risk of expired certificates causing outages or security gaps.
- CRL and OCSP revocation provides the ability to cut off unauthorized access within minutes when an agent identity is affected.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Authentication for agents has two distinct jobs: proving the agent is who it claims to be, and proving that to the receiver cryptographically rather than through network-path heuristics. Static credentials fail both tests. They are trivial to copy, hard to rotate, and their holder is indistinguishable from their issuer. The design pattern is cryptographic identity managed centrally, with lifecycle automation and revocation as first-class operations rather than afterthoughts.

[Amazon Bedrock AgentCore Identity](#) handles the OAuth side of that pattern. It provides managed OAuth 2.0 flows and identity federation for agentic workloads, issuing, validating, and rotating tokens without the operational burden of running that infrastructure. Each agent registers in the centralized agent identity directory and receives a unique identity. The `GetWorkloadAccessTokenForJWT` API issues agent-specific access tokens bound to the requesting agent's identity. The token vault secures OAuth tokens with AWS KMS encryption (customer-managed keys supported) and enforces per-agent access controls so one agent can't retrieve another's tokens.

For services that require certificate-based authentication rather than OAuth, AWS Private Certificate Authority (AWS Private CA) is the managed path for issuing internal mTLS client and server certificates. AWS Private CA handles issuance and supports automated renewal lifecycles, and certificate revocation through CRL or OCSP provides the cutoff when an identity is affected.

Mutual TLS (mTLS) on AWS Application Load Balancers or Amazon API Gateway configurations gives agent-to-agent traffic symmetric proof: both sides present certificates, both sides verify. Private keys live in AWS Secrets Manager or Parameter Store, a capability of AWS Systems Manager with AWS KMS encryption at rest and automatic rotation policies, so the key itself never becomes a long-lived static credential.

Detection rounds out the pattern. Amazon GuardDuty flags unusual authentication patterns, agents authenticating from unexpected IP addresses or at unusual times, and findings route into AWS Security Hub CSPM for centralized event management. That gives the security team a single place to see when an identity is being used in ways that don't match its normal profile, whether the credentials themselves have been revoked.

Implementation steps

1. **Deploy AgentCore Identity:** Deploy [Amazon Bedrock AgentCore Identity](#) for agent authentication, configure identity federation for cross-service access, and register each agent in the centralized identity directory for unique, trackable identities.
2. **Secure tokens in the vault:** Configure the AgentCore Identity token vault for OAuth token storage using customer-managed AWS KMS keys for encryption, and enforce strict per-agent access controls for credential retrieval.
3. **Issue agent certificates from AWS Private CA:** Set up AWS Private Certificate Authority for agent identity certificates, with automated renewal lifecycles configured.
4. **Enforce mutual TLS end-to-end:** Configure mTLS on all agent-to-agent communication endpoints through Application Load Balancers or Amazon API Gateway with mTLS authentication.
5. **Store keys in Secrets Manager with rotation:** Store all agent private keys and credentials in AWS Secrets Manager with encryption at rest and automatic rotation policies enabled.
6. **Turn on revocation checking:** Implement certificate revocation through CRL or OCSP so affected agent certificates can be invalidated immediately.
7. **Alarm on authentication anomalies:** Configure Amazon GuardDuty to detect unusual authentication patterns and route findings to AWS Security Hub CSPM for centralized security event management.
8. **Review certificate inventory quarterly:** Identify and remediate certificates approaching expiration or using deprecated algorithms on a quarterly cadence.

Resources

Related best practices:

- [AGENTSEC03-BP02 Separate agent and human user permission](#)
- [AGENTSEC03-BP03 Implement least privilege with dynamic boundaries](#)
- [AGENTSEC06-BP01 Encrypt and sign inter-agent messages](#)

Related documents:

- [Securing AI agents with Amazon Bedrock AgentCore Identity](#)
- [Amazon Bedrock AgentCore Identity documentation](#)
- [AgentCore Identity supported authentication patterns](#)
- [AWS Private CA documentation](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Private CA](#)
- [AWS Secrets Manager](#)
- [Amazon GuardDuty](#)
- [AWS Security Hub CSPM](#)

AGENTSEC03-BP02 Separate agent and human user permission

Identity propagation alone isn't enough when agents act on behalf of users. Distinct agent service identities, with hard trust boundaries helping prevent assumption of human-designated roles, keep audit attribution unambiguous and stop a misconfigured agent from escalating into a human identity space.

Desired outcome:

- Agent actions and human actions are distinguishable in audit logs, with separate identities that help prevent agents from inheriting or assuming human user permissions.

- Dedicated agent service identities are scoped to only the permissions required for automated operations, and audit logs attribute every action unambiguously to either an agent or a human actor.
- Trust policies help prevent agents from assuming human-designated identities, with organization-level guardrails providing a secondary boundary in multi-account environments.

Common anti-patterns:

- Reusing human user IAM credentials or roles for agent authentication, making it impossible to distinguish agent actions from human actions in audit logs.
- Allowing agents to assume human IAM roles through role chaining, enabling access to resources that should be restricted to human users and creating a privilege-escalation path.
- Using a single shared service account for multiple agents, reducing the risk of attribution of specific actions to individual agents and complicating incident response.
- Relying only on agent-level IAM policies without role trust policy restrictions or multi-account guardrails, leaving open the assumption path where an affected agent role could still assume a human role.

Benefits of establishing this best practice:

- Separate identities produce unambiguous audit attribution that clearly distinguishes agent actions from human actions.
- Trust policies refuse assumption by agent principals, with optional organization-level guardrails adding a multi-account ceiling.
- Consistent agent identity tagging and naming conventions enable filtering and analysis of agent activity patterns during incident investigations.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Two separations matter here and they are often collapsed in practice. Identity separation means the agent operates under a service identity distinct from any human identity, whether that human is an operator managing AWS resources or an application user interacting with the agent's service. Permission separation means the agent's identity carries only the permissions it needs for its tasks and never inherits the permissions of human identities. These apply to both patterns (agents acting

autonomously and agents acting on behalf of a user), but the way user context flows through a call differs: autonomous agents operate under their service identity alone, while agents acting on behalf of a user carry that user's context forward as token claims without the agent ever assuming the user's credentials. Human operator identities (which authenticate through IAM Identity Center for AWS access) and application user identities (which authenticate through a separate customer-facing identity provider and have no AWS identity at all) are themselves distinct layers.

For agents that operate on AWS, create dedicated IAM roles for each agent type as the agent's service identity, with a naming convention (for example, `agent-role-<agent-name>`) that distinguishes them from human operator roles, and apply `PrincipalType: Agent` as a consistent tag. Principal tags surface in AWS CloudTrail events primarily for management-plane API calls made with session tags. For service-specific events that don't surface principal tags, filter by the agent role ARN naming convention as the primary signal and treat the tag as a secondary filter. Use AWS IAM Identity Center for all human operator access and migrate any human operators still using account-specific IAM users off those identities. Application users are typically outside this migration, they authenticate through the application's own identity provider, and their identity flows into agent calls as token claims. Configure Service Control Policies (SCPs) in AWS Organizations that explicitly deny agents (identified by tag) from assuming roles in the human operator identity boundary, for example denying `sts:AssumeRole` for any principal tagged as an agent attempting to assume roles tagged as human-operator.

Amazon Bedrock AgentCore introduces two identity constructs that support this separation. The AgentCore Runtime **execution role** is a regular IAM role assumed by the Runtime process to reach AWS services the agent depends on (model invocations, tool calls to AWS APIs, memory reads and writes). The AgentCore **Workload Identity** is an agent-specific identity registered in the AgentCore Identity directory, distinct from any IAM role, and used to issue and verify agent-scoped tokens and vault third-party tokens obtained on behalf of users. When an agent acts on behalf of an application user, it calls `GetWorkloadAccessTokenForJWT` with the user's access token from the application's identity provider. [Amazon Bedrock AgentCore Identity](#) verifies the user token and returns an agent-specific workload access token that embeds the user context as claims, so the call chain carries both "who the agent is" and "who the user is" without the agent ever assuming the user's credentials or an IAM role on the user's behalf.

For agents that need to reach third-party OAuth resources on behalf of users, AgentCore Identity orchestrates the authorization code grant flow and secures the resulting access tokens in a token vault scoped per agent identity and per user. One agent can't retrieve tokens obtained for a different user, and as long as the third-party access token remains valid the agent can retrieve it from the vault without requiring the user to re-authenticate.

Amazon Cognito provides an alternative for organizations already standardized on Cognito user pools. The same principle (the agent carries both its own identity and the user's context as token claims) can be implemented outside AgentCore Identity. The pattern described in [Empower AI agents with user context using Amazon Cognito](#) uses [Amazon Cognito](#) user pools: the agent authenticates through the client credentials flow while passing the user's token as additional context through the `aws_client_metadata` request parameter, and a [pre-token-generation Lambda trigger](#) verifies the user token and injects an `onBehalfOf` claim into the agent's access token before issuance. The resulting JWT identifies the agent in its `sub` claim and the user in its `onBehalfOf` claim, producing a single token that downstream authorization layers can evaluate. This is a custom extension built on Cognito trigger hooks, not a native Cognito feature, and it complements the AgentCore pattern rather than replacing it. For authorization on top of the resulting token, resource servers can evaluate claims directly, or externalize policy to a service such as [Amazon Verified Permissions](#) using Cedar (a choice orthogonal to whether you use AgentCore Identity or the Cognito pattern).

When an agent invokes a tool through AgentCore Gateway, the gateway operates under its own execution role (one per gateway) and the outbound authentication applied to the target depends on the target type (see the [AgentCore Gateway outbound auth reference](#) for details). User context propagates through the gateway so downstream services can enforce user-level authorization, and the gateway never substitutes an IAM identity for the user or re-authenticates as the user against downstream services.

Configure [CloudTrail with advanced event selectors](#) to capture agent API calls, including data events for agent-invoked services where visibility beyond management-plane events matters. Enable AgentCore data events so operations against AgentCore Memory, Gateway, and Runtime are logged alongside management-plane events. Amazon CloudWatch Logs Insights queries filtered by agent role ARN patterns give dedicated agent activity dashboards. For SQL-based analysis over time, deliver CloudTrail logs to Amazon S3 and query them with Amazon Athena (see AGENTSEC05-BP01).

For multi-agent systems where a parent agent directly assumes a sub-agent's IAM role (for example, calling `sts:AssumeRole` before invoking the sub-agent in the same account), use role session names that include the parent agent's identifier so the chain is visible in CloudTrail. For A2A communication that routes through AgentCore Runtime, each agent runs under its own independent execution role session and there is no IAM role chain. Use correlation identifiers propagated through the A2A message (see AGENTSEC05-BP02) to reconstruct the agent-to-agent call graph.

Implementation steps

1. **Audit and rename agent roles:** Identify any IAM roles shared between agents and human users, and create dedicated agent-specific roles with a consistent naming convention (for example, agent-role-<agent-name>).
2. **Tag agent roles consistently:** Apply PrincipalType: Agent and AgentName: <name> tags to every agent IAM role for filtering and monitoring.
3. **Move human operators to IAM Identity Center:** Configure AWS IAM Identity Center for all human operator access and migrate any operators still on account-specific IAM users.
4. **Enforce the agent-to-human boundary through SCPs:** Deploy Service Control Policies in AWS Organizations that deny agents (identified by tag) from assuming human-operator roles, creating a hard boundary between the identity spaces.
5. **Register Workload Identities for on-behalf-of flows:** Register agent Workload Identities in [Amazon Bedrock AgentCore Identity](#) for agents that act on behalf of users, and use GetWorkloadAccessTokenForJWT to create agent-scoped tokens that embed user context as claims for downstream authorization.
6. **Enable CloudTrail with AgentCore data events:** Turn on advanced event selectors including AgentCore data events, and build Amazon CloudWatch Logs Insights queries filtered by agent role ARN patterns for dedicated agent activity dashboards.
7. **Analyze with Athena over time:** Deliver AWS CloudTrail logs to Amazon S3 and use Amazon Athena for SQL-based analysis of agent compared to human activity patterns over time for compliance reporting and long-term trend analysis.
8. **Name sessions for attribution:** Implement role session naming conventions that include agent identifiers (and parent agent identifiers for delegation chains) for clear attribution in multi-agent systems.

Resources

Related best practices:

- [AGENTSEC03-BP01 Implement strong authentication for agent identities](#)
- [AGENTSEC03-BP03 Implement least privilege with dynamic boundaries](#)
- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)

Related documents:

- [AWS CloudTrail best practices](#)
- [AWS IAM Identity Center](#)
- [Securing AI agents with Amazon Bedrock AgentCore Identity](#)
- [Empower AI agents with user context using Amazon Cognito](#)
- [Amazon Bedrock AgentCore Identity documentation](#)
- [AgentCore Identity supported authentication patterns](#)

Related services:

- [AWS Identity and Access Management](#)
- [AWS IAM Identity Center](#)
- [AWS CloudTrail](#)
- [AWS Organizations](#)
- [Amazon CloudWatch](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon Cognito](#)
- [Amazon Verified Permissions](#)

AGENTSEC03-BP03 Implement least privilege with dynamic boundaries

Broad permissions grant an affected agent access well beyond the task it was asked to do. Scoping privilege at each identity layer, backing it with temporary credentials, and layering contextual IAM conditions limits the scope of any single compromised or misprompted call.

Desired outcome:

- Agents operate with the minimum permissions required to complete their defined tasks, with temporary credentials that expire automatically and dynamic permission boundaries that tighten when handling sensitive data or high-risk operations.
- Just-in-time access patterns help limit elevated permissions to the duration of the operation that requires them.
- You continually validate policy scoping as workloads evolve, removing drift before it accumulates.

Common anti-patterns:

- Assigning broad IAM policies (for example, s3:* or *) to agent roles for convenience, granting far more access than any individual task requires.
- Using long-lived static credentials instead of temporary credentials from AWS STS, extending the window of exposure if credentials are inadvertently disclosed.
- Failing to implement IAM permission boundaries, allowing agents to create or modify IAM policies and escalate their own privileges.
- Expanding agent permissions in response to access errors without investigating whether the access pattern is legitimate, accumulating excessive permissions through reactive grants that are never revoked.

Benefits of establishing this best practice:

- Least-privilege IAM roles limit an affected agent to only the resources required for its current task.
- Temporary AWS STS credentials with short session durations reduce the risk of long-lived credential exposure.
- IAM Conditions on region, resource tags, time windows, and source network add defense-in-depth that limits the impact of credential exposure.

Level of risk exposed if this best practice is not established: High

Implementation guidance

An agent operates under multiple identity layers, and each is a separate place to apply scoping. The **agent identity** is what this specific agent is as an entity in your directory (for example, an AgentCore Workload Identity with a unique ARN). The **service identity** is what the agent runs as when it invokes AWS services (for AgentCore Runtime. This is the IAM execution role assumed by the Runtime process). The **transaction identity** is the per-invocation context carried with a single agent call, typically expressed through AWS STS session credentials with session tags and session policies. When an agent acts on behalf of a user, a **user identity** is additionally carried in the call context as token claims propagated through the agent's call chain. Permission scoping applies differently at each layer:

- Broad capability limits belong on the service identity (IAM role, permission boundary)

- Per-operation constraints belong on the transaction (session policy, session tags, and IAM Conditions)
- User-context constraints belong on the user identity (token-based authorization evaluated at the resource)

At the service identity layer, design agent IAM roles using the principle of least privilege, starting with no permissions and adding only what is required for each specific task. AWS IAM Access Analyzer generates least-privilege policies based on actual access patterns observed in AWS CloudTrail logs, giving you a data-driven baseline for scoping. IAM permission boundaries on all agent roles establish a maximum permission ceiling that can't be exceeded even if the agent's policies are modified, which matters because it is the control that helps prevent privilege escalation when something about the policy itself changes.

[Amazon Bedrock AgentCore Gateway](#) operates under a single execution role per gateway, with outbound authentication that depends on target type (see the [AgentCore Gateway outbound auth reference](#)). Scope the gateway execution role to only the actions and resources the set of targets collectively requires, and constrain per-target access through the target-specific outbound auth configuration. User context propagated through the gateway lets downstream services enforce user-level access controls in addition to the gateway-level role, so operations are further constrained by the originating user's permissions even if the gateway role has access to a target.

Temporary credentials are how the transaction layer stays scoped. AWS STS AssumeRole with session policies generates credentials scoped to the specific permissions required for each task execution, and short session durations (15 to 60 minutes) make expiration automatic. For agents that require elevated permissions for specific operations, just-in-time access through AWS IAM Identity Center or custom Lambda-based access request workflows grants and revokes permissions programmatically.

IAM Conditions are the defense-in-depth layer that makes exposed credentials less useful. `aws:RequestedRegion` limits agent actions to approved regions. `aws:ResourceTag` restricts access to resources tagged for agent use. `aws:CurrentTime` enforces time-window restrictions. `aws:SourceVpc` requires that agent API calls originate from approved VPCs. These conditions don't reduce the policy's stated permissions, but they bound the contexts under which those permissions apply. At the organization level, Service Control Policies in AWS Organizations establish organization-wide guardrails that apply to all agent accounts, helping prevent agents from accessing services or performing actions that are never appropriate regardless of task-level permissions.

Implementation steps

1. **Generate least-privilege policies from usage data:** Audit existing agent IAM roles with AWS IAM Access Analyzer to identify overly permissive policies and generate least-privilege recommendations based on actual access patterns in AWS CloudTrail logs.
2. **Apply permission boundaries:** Set IAM permission boundaries on all agent roles to establish a maximum permission ceiling that can't be exceeded even if the agent's task-level policies are modified.
3. **Scope the Gateway execution role:** Restrict the [Amazon Bedrock AgentCore Gateway](#) execution role to only the actions and resources its targets require, configure target-specific outbound auth per target type, and use AgentCore Identity to propagate user context for downstream user-level enforcement.
4. **Issue temporary credentials with session policies:** Use AWS STS AssumeRole with session policies and short session durations (15 to 60 minutes) for all agent credential issuance.
5. **Add contextual IAM Conditions:** Restrict access by region (aws:RequestedRegion), resource tags (aws:ResourceTag), time windows (aws:CurrentTime), and source network (aws:SourceVpc).
6. **Deploy organization-wide SCPs:** Establish organization-wide guardrails through Service Control Policies in AWS Organizations that apply to all agent accounts.
7. **Implement just-in-time access for elevated permissions:** Use IAM Identity Center or custom Lambda-based access request workflows with automatic revocation when the operation completes or the session expires.
8. **Review IAM drift quarterly:** Schedule IAM Access Analyzer reviews every quarter to detect permission drift, identify unused access, and remove permissions that are no longer required.

Resources

Related best practices:

- [AGENTSEC02-BP01 Implement tool authorization](#)
- [AGENTSEC03-BP01 Implement strong authentication for agent identities](#)
- [AGENTSEC03-BP02 Separate agent and human user permission](#)
- [AGENTSEC03-BP04 Regular permission audits and access reviews](#)
- [AGENTREL02-BP02 Limit agent permissions to minimum required access](#)

Related documents:

- [AWS IAM best practices](#)
- [AWS IAM Access Analyzer](#)
- [Amazon Bedrock AgentCore Identity documentation](#)
- [Amazon Bedrock AgentCore Gateway documentation](#)
- [Service control policies](#)

Related services:

- [AWS Identity and Access Management](#)
- [AWS IAM Access Analyzer](#)
- [AWS Organizations](#)
- [Amazon Bedrock AgentCore](#)

AGENTSEC03-BP04 Regular permission audits and access reviews

Agent roles accumulate permissions over time. Scheduled automated analysis paired with periodic human-led reviews catches drift before it turns into significant over-privilege and produces the documented audit trail that compliance needs.

Desired outcome:

- You continually monitor agent permissions and review them on a cadence matched to the agent's risk profile, identifying and removing unused access regularly.
- Automated alerts fire immediately when agent permissions are modified, enabling rapid detection of unauthorized policy changes.
- You document access reviews with timestamped findings and remediation actions for compliance purposes.

Common anti-patterns:

- Conducting permission reviews only annually or in response to incidents, letting drift accumulate undetected for months.
- Setting a single review cadence for every agent regardless of risk, so high-risk agents (those with broad permissions, mutating tool access, or production data reach) receive the same scrutiny as low-risk informational agents.

- Reviewing permissions manually without tooling support, making it impractical to assess the full scope of agent access across dozens or hundreds of roles.
- Treating IAM Access Analyzer findings as informational rather than as practical remediation items, so identified over-privilege persists indefinitely.
- Not alerting on permission changes in real time, discovering unauthorized policy modifications only during the next scheduled review cycle weeks or months later.

Benefits of establishing this best practice:

- Ongoing permission monitoring detects drift before it accumulates into significant over-privilege.
- Timestamped findings and remediation actions support compliance requirements and security investigations.
- Usage-based evidence from AWS CloudTrail drives permission reduction with data rather than guesswork about which permissions are still needed.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Two review modes are both necessary. Automated analysis provides speed and coverage, the ability to scan every agent role across every account every day. Periodic human-led reviews provide the context that automation can't supply, such as whether a technically unused permission is still needed for upcoming work, whether a recent policy change was expected, and whether the current privilege level matches the current role of the agent in the business. Running only one of the two leaves a gap: automation alone produces findings no one acts on, and manual-only reviews happen too infrequently to catch drift in time.

AWS IAM Access Analyzer at the organization level continually analyzes agent IAM policies and generates findings for permissions that grant access to resources outside the intended scope. Its unused-permissions analysis uses AWS CloudTrail activity data to identify access that has not been exercised, giving a data-driven basis for permission reduction rather than a guess. Weekly review and remediation of Access Analyzer findings, prioritized by severity, keeps the backlog bounded and turns findings into change tickets.

AWS Config rules detect changes to agent IAM policies, roles, and permission boundaries in near real time. Configure managed rules such as iam-policy-no-statements-with-admin-access along

with custom rules that validate agent-specific policy constraints, and route rule violations through Amazon EventBridge to an Amazon SNS topic so the security team is notified immediately rather than during the next scheduled review.

For the formal periodic review, correlate Access Analyzer findings with CloudTrail usage data to identify permissions that have not been exercised in the period. The review cadence should match the risk profile of the agent: high-risk agents (those with broad permissions, write access to production data, or mutating tool privileges) warrant frequent review, while low-risk informational agents can be reviewed less often. [AWS CloudTrail Lake](#) provides queryable long-term retention for this analysis, and AWS Lambda functions can automate the generation of review reports by querying IAM and CloudTrail data and publishing results to Amazon S3. The output feeds the review meeting and produces the documented evidence compliance requires.

AWS Security Hub CSPM is the aggregation layer for large agent fleets. Findings from IAM Access Analyzer, AWS Config, and Amazon GuardDuty flow into a single view where severity and business impact drive prioritization, so the team is working from one list instead of three consoles.

Implementation steps

1. **Enable organization-level IAM Access Analyzer:** Turn on AWS IAM Access Analyzer at the organization level and configure it to analyze all agent IAM roles for unused and overly permissive access.
2. **Detect policy changes with AWS Config:** Deploy AWS Config rules to detect changes to agent IAM policies and trigger Amazon EventBridge notifications for immediate alerting to the security team.
3. **Retain activity data in CloudTrail Lake:** Configure AWS CloudTrail Lake for long-term retention of agent API activity data, supporting access review correlation and compliance reporting.
4. **Automate weekly finding reviews:** Implement automated weekly reviews of Access Analyzer findings, generating reports that prioritize high-severity findings for remediation.
5. **Run a formal access review on a risk-based cadence:** Set the review cadence per agent based on its risk profile (high-risk agents reviewed frequently, low-risk informational agents reviewed less often). Correlate Access Analyzer findings with CloudTrail usage data, document findings and remediation actions, and record sign-off for each review cycle.
6. **Aggregate findings in Security Hub CSPM:** Pull IAM findings from Access Analyzer, AWS Config, and Amazon GuardDuty into AWS Security Hub CSPM for a unified view of permission-related issues across the agent fleet.

Resources

Related best practices:

- [AGENTSEC03-BP03 Implement least privilege with dynamic boundaries](#)
- [AGENTSEC03-BP02 Separate agent and human user permission](#)
- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)

Related documents:

- [AWS IAM Access Analyzer documentation](#)
- [AWS Config managed rules](#)

Related services:

- [AWS IAM Access Analyzer](#)
- [AWS Config](#)
- [Amazon EventBridge](#)
- [AWS CloudTrail](#)
- [AWS Security Hub CSPM](#)
- [Amazon CloudWatch](#)

Agent goal alignment and manipulation prevention

Agents can be directed to pursue unintended objectives or exhibit behaviors outside their defined scope. Without goal alignment mechanisms, agents might take actions that conflict with organizational policies or user intent. Implementing guardrails, layered validation controls, and human-in-the-loop approvals helps verify that agents operate within intended boundaries.

AGENTSEC04: How do you support agent goal alignment and prevent manipulation?

Capability intent

- Operational and policy boundaries for each agent are defined up front and enforced through layered controls rather than prompt instructions alone.
- Deterministic enforcement (IAM scoping, input schema validation, and policy engines) and probabilistic content controls (input and output filtering, behavioral evaluation) operate at distinct stages of the call chain, so a failure at one layer rarely results in a boundary violation.
- Risk classification is deterministic, with high-risk operations routed to human review before execution and routine low-risk operations proceeding autonomously.
- Reviewers receive sufficient decision context, timeout policies, and escalation paths to make informed approvals without stalling the workflow or defaulting to rubber-stamping.
- Guardrail interventions, approval decisions, and evaluation results are logged, alerted on, and reviewed on a defined cadence to keep policies current with emerging patterns.

Maturity levels

These levels summarize what each stage of maturity looks like for agent goal alignment and manipulation prevention as a whole.

Level	Name	What it looks like
1	Initial	Operational boundaries exist only as natural-language instructions in the system prompt. No content guardrails, input validation, or human-in-the-loop controls are in place. Alignment depends on the model following instructions, and a single adversarial input can redirect the agent with no defense in depth.
2	Emerging	Amazon Bedrock Guardrails are configured with basic content filters covering inputs and outputs. A

Level	Name	What it looks like
		single approval tier exists for a subset of consequential actions, implemented through built-in mechanisms such as Amazon Bedrock Agents user confirmation or routed through ad-hoc channels such as email or chat. Guardrail interventions are captured in logs but reviewed irregularly.
3	Defined	Tiered guardrail configurations align filter strengths to each agent's use case, and denied topics and word filters are calibrated per risk profile. Deterministic risk classification routes approvals through a structured workflow mechanism such as AWS Step Functions callback workflows or return of control , with defined timeouts and escalation paths. Amazon CloudWatch alarms fire on intervention spikes, and guardrail versioning supports rollback of policy changes.

Level	Name	What it looks like
4	Proactive	Validation runs at every stage of the call chain: input, inference, output, and tool invocation. Amazon Bedrock AgentCore Evaluations run on a defined cadence with alarms on correctness, tool selection accuracy, and safety score drift. Persistent trust grants are scoped narrowly by command and parameter shape, tiered by risk, and made auditable and revocable . Full decision context is written to durable storage before approval notifications, and approval records are linked to the centralized audit trail.

Level	Name	What it looks like
5	Optimized	Alignment controls are calibrated from intervention and evaluation data, and risk classification adapts to dynamic signals such as operation frequency, time of day, and source location. Multi-reviewer consensus is available for the highest-risk operations, including asynchronous workflows running on Amazon Bedrock AgentCore Runtime . Policy updates flow through versioned configurations with provable change history, and the feedback loop from logged interventions back into policy is automated rather than manual.

Common issues to watch for

- Alignment is treated as a prompt-engineering exercise rather than a layered enforcement problem, so a single adversarial input that influences the model can collapse multiple boundaries at once.
- Content filtering is applied only to model outputs, leaving the inference path open to prompt injection and consuming capacity on adversarial inputs that could have been rejected up front.
- A single guardrail profile is applied to every agent, which either over-constrains low-risk informational agents or leaves high-risk operational agents under-constrained.
- Approval workflows either route every action through human review (producing reviewer fatigue and rubber-stamping) or skip review for operations that warrant it, giving the worst of both extremes.

- Risk classification is delegated to an LLM exposed to the same untrusted content as the request it is evaluating, which can be influenced into marking that request as low-risk.

Best practices

- [AGENTSEC04-BP01 Implement guardrails and alignment controls](#)
- [AGENTSEC04-BP02 Human-in-the-loop for critical decisions](#)

AGENTSEC04-BP01 Implement guardrails and alignment controls

Instruction-following alone doesn't provide reliable enforcement. Layered controls (deterministic where possible, probabilistic where necessary) help keep agents inside operational boundaries even when prompts are adversarial and model behavior is unpredictable.

Desired outcome:

- You define agent operational and policy boundaries up front and enforce them through layered controls, with deterministic controls (IAM, schema validation, technical policy checks) handling what is expressible deterministically and probabilistic controls (content filters, behavioral evaluation) handling what isn't.
- Multiple validation layers at different stages of the agent call chain can reduce the likelihood that a single control failure results in a boundary violation.
- You log, alert on, and periodically review guardrail interventions, policy violations, and behavioral evaluation results to tune the controls and surface emerging patterns.

Common anti-patterns:

- Relying on a single guardrail configuration for all agent use cases, applying the same constraints to low-risk informational agents and high-risk operational agents.
- Applying content filtering only to model outputs without validating inputs first, letting adversarial content reach the model before any check runs.
- Defining operational boundaries in natural-language system prompts alone, relying on the model's instruction-following as the sole constraint, which can be bypassed through prompt injection or adversarial framing.

Benefits of establishing this best practice:

- Deterministic technical controls (IAM, schema validation) combined with probabilistic content controls (Guardrails) at distinct stages reduce reliance on instruction-following alone.
- Layered validation catches policy violations at multiple stages, so a bypass at one layer is less likely to result in an unchecked boundary violation.
- Logged guardrail interventions and evaluation results feed policy updates as new patterns emerge, keeping boundaries current with evolving use cases.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Operational boundaries written only in system prompts typically don't provide reliable enforcement. A prompt can be overridden by adversarial framing, prompt injection, or the model's own creative reinterpretation, and none of those failure modes produces an audit signal before the boundary has already been crossed. The design pattern is layered. Express what can be expressed deterministically as hard checks:

- IAM scoping
- Schema validation
- Cedar policies
- Permission boundaries

Use probabilistic controls (content filters, behavioral evaluation) to cover the content-shaped risks that determinism can't reach.

[Amazon Bedrock Guardrails](#) is the probabilistic layer. Configure a base guardrail with universal constraints (no generation of harmful content, no disclosure of system prompts), then overlay use-case-specific configurations for each agent's operational context. Content filter strengths need calibration to use case sensitivity: HIGH strength for consumer-facing agents handling categories like hate speech, violence, and sexual content, MEDIUM strength for internal enterprise agents, and custom thresholds for specialized domains (medical, legal) with their own content norms. Content moderation needs to apply to every output path, including outputs used in internal agent workflows and inter-agent messages, not just user-facing responses. Use Guardrails versioning for change management with rollback.

Pre-execution validation matters for two reasons. Applying Guardrails to user inputs before they reach the model blocks adversarial content before it influences reasoning, and it rejects bad inputs

before they consume inference capacity. When an agent invokes an Amazon Bedrock model with a guardrail attached, the check runs automatically on inputs and outputs. The ApplyGuardrail API runs the same policy independently when the automatic path doesn't apply, agents that invoke non-Amazon Bedrock models (third-party APIs, self-hosted models), pipelines that need to filter content before deciding whether to invoke a model at all (for example, checking retrieved content before it enters the prompt), or additional validation checkpoints beyond the model invocation.

Monitoring closes the feedback loop. The [Amazon Bedrock Guardrails CloudWatch metrics](#) include `InvocationsIntervened`, broken down by `Operation` (ApplyGuardrail) and `GuardrailContentSource` (Input / Output) so input-side and output-side interventions are visible separately. Amazon CloudWatch alarms on intervention rates route through Amazon SNS to the security team. For the detail (what was blocked, which policy triggered, which part of the content was affected), enable Amazon Bedrock model invocation logging, which captures the full guardrail trace for each call. Analyze intervention patterns over time to find emerging techniques that require policy updates and to catch filter categories generating excessive false positives or false negatives.

[Amazon Bedrock AgentCore Evaluations](#) assesses goal attainment correctness and tracks whether agents are achieving their intended objectives rather than drifting into misaligned goals. Built-in evaluators cover correctness, helpfulness, tool selection accuracy, and safety. Custom model-based evaluators extend coverage to organization-specific alignment requirements. Run evaluations on a regular cadence and after any significant change to agent prompts, tools, or guardrail configurations. Results publish to Amazon CloudWatch alongside AgentCore Observability insights for a unified view, and CloudWatch alarms on evaluation scores catch behavioral drift outside acceptable thresholds.

Implementation steps

1. **Map ethical constraints to guardrail categories:** Define organizational ethical constraints and operational boundaries per agent use case and map them to guardrail policy categories with risk-appropriate differentiation.
2. **Build tiered guardrail configurations:** Create base and use-case-specific [Amazon Bedrock Guardrails](#) configurations, apply them to all deployments, and use versioning for rollback capability.
3. **Validate inputs before inference:** Call the ApplyGuardrail API on inputs before they reach the model, checking against denied topics, word filters, and sensitive information patterns.
4. **Alarm on intervention metrics:** Configure Amazon CloudWatch alarms on Guardrails metrics (especially `InvocationsIntervened`), route alerts through Amazon SNS, and enable Amazon Bedrock model invocation logging for detailed intervention records.

5. **Deploy AgentCore Evaluations with drift alarms:** Deploy [Amazon Bedrock AgentCore Evaluations](#) with built-in and custom evaluators, and configure Amazon CloudWatch alarms on evaluation scores to detect behavioral drift.
6. **Review intervention logs monthly:** Establish a monthly review of guardrail intervention logs to identify emerging patterns and update policies accordingly.

Resources

Related best practices:

- [AGENTSEC04-BP02 Human-in-the-loop for critical decisions](#)
- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)
- [AGENTSEC08-BP01 Multi-layer input validation and prompt injection defense](#)

Related documents:

- [Amazon Bedrock Guardrails documentation](#)
- [Amazon Bedrock Guardrails content filters](#)
- [Build responsible AI applications with Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore Evaluations](#)
- [Amazon Bedrock AgentCore adds quality evaluations and policy controls](#)
- [AI agents in enterprises: Best practices with Amazon Bedrock AgentCore](#)

Related services:

- [Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon EventBridge](#)
- [Amazon CloudWatch](#)
- [Amazon SNS](#)

AGENTSEC04-BP02 Human-in-the-loop for critical decisions

Routing every agent action through human review produces rubber-stamp approvals. Routing none produces unbounded autonomy. Risk-tiered approval pauses agents only for the decisions where human judgment actually changes the outcome, with enough context to make those decisions meaningful.

Desired outcome:

- You pause high-risk agent operations for human review before execution, and reviewers receive enough context to make informed decisions within a defined time window.
- Escalation paths handle cases where primary reviewers are unavailable.
- You log human approval decisions with timestamps and reviewer identities, creating an auditable record of human oversight for compliance purposes.

Common anti-patterns:

- Routing all agent actions through human review regardless of risk level, creating reviewer fatigue and rubber-stamp approvals.
- Providing reviewers with insufficient context (the proposed action only, without the reasoning chain, data sources, or potential consequences), turning review into a formality.
- Implementing approval workflows without timeout policies or escalation paths, so agent execution stalls indefinitely when reviewers are unavailable.

Benefits of establishing this best practice:

- Risk-tiered approval workflows focus reviewer attention on the decisions where human judgment matters most.
- Logged approval decisions with reviewer identity and timestamps produce an auditable record that satisfies compliance requirements.

Level of risk exposed if this best practice is not established: High

Implementation guidance

A risk classification framework is what lets human attention go where it adds value. Combine static properties of the operation (what kind of action, against what resource) with dynamic signals

about the request (frequency, time of day, source location, recent anomalies). Risk classification itself can't rely on an LLM exposed to the same untrusted content as the request being evaluated, because adversarial content could influence the classifier into marking the request as low-risk. Use deterministic logic (policy engines, rule-based classifiers) as the authoritative signal, with LLM-assisted classification as an optional input that a deterministic layer re-checks. As a baseline: read-only operations proceed autonomously, low-risk writes require single-reviewer approval, and higher-risk operations (financial transactions, data deletion, external communications) require stricter approval, which can be single-reviewer, multi-reviewer, or out-of-band depending on your risk policy. For established risk framing approaches to adapt, see the [AWS Agentic AI Security Scoping Matrix](#) and [this guide to building AI agents in GxP environments](#).

Persistent trust grants, where a reviewer approves a specific operation pattern once and future operations matching the pattern proceed without re-approval, are a useful escape valve for genuinely routine operations, but they shift where human judgment is applied from moment-to-moment to at-grant time. If you implement persistent trust, bound each grant to a specific command, parameter shape, or resource. Tier grants by risk so higher-risk operations are ineligible for persistent trust or require re-confirmation at a defined cadence, and make grants themselves auditable and revocable. Wildcard trust grants (approving all future operations of a given type with no parameter scoping) effectively remove human oversight from an entire class of operations and should not be issued.

Once the classifier says human approval is required, route the request to the approval mechanism appropriate for the agent's execution environment. Three patterns cover most deployments. For agents embedded in step-function-driven workloads, [AWS Step Functions](#) .waitForTaskToken callback pattern introduces an approval step. The workflow emits a task token and pauses, the token is delivered to an approval application through a channel appropriate to the reviewer population (Amazon SNS to a queue that the approval app consumes, Amazon SES to a reviewer mailbox, or a webhook endpoint on the approval app), the application presents the decision to the reviewer, and the application calls SendTaskSuccess or SendTaskFailure with the task token on the reviewer's behalf. Reviewers don't typically call Step Functions APIs directly. The approval app holds the credentials, and the reviewer interacts with the app. See the [human approval in Step Functions tutorial](#) for a worked example.

For agents built on [Amazon Bedrock Agents](#), two built-in patterns handle human-in-the-loop confirmation without external workflow orchestration. [User confirmation](#) pauses the agent before executing a specific function and returns the function name and parameter values to the calling application for yes/no presentation to the user. [Return of control \(ROC\)](#) goes further by returning the function call itself so the application decides what to do (present to a user, run validation,

modify parameters, or reject). ROC is configured at the action group level and covers all actions in that group. Both patterns assume the application is the component implementing the human approval UX. These are better suited for interactive use cases where the end user of the application is also the approver, while Step Functions callback patterns fit asynchronous workflows with separate reviewer roles.

For agents that need long-running approval processing, [Amazon Bedrock AgentCore Runtime](#) supports both synchronous and asynchronous processing through a unified API, enabling an agent to start a task that may take minutes or hours, immediately acknowledge the request, continue approval workflows in the background, and let the user check back later for results. The SDK provides explicit task lifecycle management through `add_async_task` and `complete_async_task` APIs, which track processing status and report agent health through the `/ping` endpoint. An agent reports `HealthyBusy` while background approval tasks are in progress and `Healthy` when idle. This is particularly useful for multi-reviewer consensus workflows where approval collection spans extended periods. Blocking operations such as waiting for reviewer responses need to run in separate threads or use async methods to avoid blocking the health-check endpoint, which would cause the runtime session to terminate after 15 minutes of unresponsive pings.

Reviewers need enough context to make informed decisions without wading through raw logs. Decisions involving agent reasoning often produce a large volume of intermediate content (prompts, tool outputs, retrieved documents, model responses) that is too much to deliver in a notification payload and may be accessed by reviewers who are not signed into the agent's application. Store the full decision context in durable storage such as Amazon S3 before sending the approval notification, including the agent's reasoning chain, the proposed action, relevant data sources, and potential consequences. Make the context available through the same authenticated interface the reviewer uses to approve or deny. When reviewers don't have access to the approval system's UI (for example, approvals through email), presigned S3 URLs with short expiration times provide temporary access to the context document. Structure the context to highlight the key decision factors.

Timeout policies and escalation paths are specific to each approval mechanism. In Step Functions, `TimeoutSeconds` or `HeartbeatSeconds` on the task state waiting for the approval token triggers a timeout transition, and `Catch` clauses route timed-out executions to an escalation state (notify secondary reviewers, escalate to management, or default to a safe fallback, typically blocking the operation). See [Step Functions error handling](#) for the attribute details. For Amazon Bedrock Agents user-confirmation and ROC flows, timeouts and escalation are handled by the calling application. For AgentCore Runtime async tasks, the underlying Step Functions timeouts apply because the async task typically waits on a Step Functions callback or equivalent external signal.

Approval decisions need to be logged for compliance and audit. Capture reviewer identity, notification and response timestamps, the operation under review, the decision, and any escalation events. Step Functions emits execution history to Amazon CloudWatch Logs when logging is enabled at the workflow level (see [Step Functions logging](#)). Augment this with application-level logs from the approval app so reviewer identity and decision rationale are captured alongside the `SendTaskSuccess/SendTaskFailure` calls. For AgentCore-based agents, the AgentCore Observability session and trace hierarchy captures agent-side events and pairs with the approval-app logs for a complete record. Retain logs consistent with your compliance requirements and AGENTSEC05-BP01.

Implementation steps

1. **Define a deterministic risk classifier:** Map agent operation types to approval tier requirements (autonomous, single-reviewer, multi-reviewer), combine static classification with dynamic request-time signals, and implement the classifier as deterministic logic rather than an LLM.
2. **Match approval mechanism to execution environment:** Use [AWS Step Functions](#) callbacks for step-function-driven agents, [Amazon Bedrock Agents](#) user confirmation or return-of-control for agents on Amazon Bedrock Agents, and [Amazon Bedrock AgentCore Runtime](#) async tasks for agents that need long-running approval processing, and implement the corresponding pattern for each tier that requires human approval.
3. **Bound any persistent trust grants:** Scope each grant narrowly (specific command, parameter shape, or resource), tier grants by risk, and make them auditable and revocable.
4. **Store decision context durably:** Write full decision context to Amazon S3 before sending approval notifications, and make the context available through the authenticated approval interface or through short-lived presigned URLs when that isn't possible.
5. **Configure timeouts with safe fallbacks:** Implement timeout policies and escalation paths for each approval mechanism, with safe fallback actions (typically blocking the operation) when no reviewer responds in the defined window.
6. **Log every approval:** Capture reviewer identity, timestamps, operation under review, decision, and escalation events, aligned with AGENTSEC05-BP01 retention and compliance requirements.
7. **Review workflow metrics periodically:** Look for patterns that suggest reviewer fatigue or process inefficiencies and adjust risk-tier thresholds accordingly.

Resources

Related best practices:

- [AGENTSEC04-BP01 Implement guardrails and alignment controls](#)
- [AGENTSEC07-BP01 Implement cognitive load management](#)
- [AGENTSEC07-BP03 Multiple reviewers for critical operations](#)

Related documents:

- [AWS Step Functions callback patterns](#)
- [Human approval in Step Functions](#)
- [Step Functions error handling](#)
- [Step Functions logging](#)
- [Implement human-in-the-loop confirmation with Amazon Bedrock Agents](#)
- [Amazon Bedrock Agents user confirmation](#)
- [Amazon Bedrock Agents return of control](#)
- [Amazon Bedrock AgentCore Runtime asynchronous processing](#)
- [AWS Agentic AI Security Scoping Matrix](#)
- [A guide to building AI agents in GxP environments](#)

Related services:

- [AWS Step Functions](#)
- [Amazon SNS](#)
- [Amazon SES](#)
- [Amazon S3](#)
- [Amazon CloudWatch](#)

Agent observability and non-repudiation

Without proper logging and traceability, agent actions can't be investigated or attributed. Agents operate autonomously across multiple services, tools, and data sources, generating complex interaction chains that are difficult to reconstruct after the fact. Observability through logging, distributed tracing, and decision artifact storage supports security investigations, compliance reporting, and continuous improvement of agent behavior.

Two decisions drive this question. First, what record of agent behavior you keep, where it lives, and how it is protected against tampering by the agent itself. Second, how you follow a single request across agents, tools, and asynchronous boundaries so that any action can be reconstructed end-to-end.

AGENTSEC05: How do you implement observability and prevent repudiation?

Capability intent

- Agent decisions, reasoning chains, tool invocations, and intermediate steps are captured to tamper-evident, queryable storage that sits outside the agent's own operational scope.
- Every logged action carries the initiating source that caused the agent to act, whether a human user session, an upstream event, a schedule, or another agent, so logged actions can typically be traced back to what triggered them.
- A single correlation identifier, independent of the tracing system's trace ID, survives asynchronous boundaries and connects every span, log, and decision artifact generated during a request.
- Sensitive fields are masked or redacted before logs reach long-term storage, so completeness of behavioral records doesn't conflict with data protection obligations.
- Security investigators can reconstruct the complete chain of agent interactions for past requests, without relying on the agent's own account of its reasoning.

Maturity levels

These levels summarize what each stage of maturity looks like for agent observability and non-repudiation as a whole.

Level	Name	What it looks like
1	Initial	Only final agent outputs are captured, typically to application logs that share credentials and lifetime with the agent itself. Intermediate reasoning, tool invocatio

Level	Name	What it looks like
		ns, and trigger attribution are lost. Logs are mutable, sensitive data is unmasked, and there is no practical way to reconstruct how the agent reached a given conclusion.
2	Emerging	Amazon Bedrock model invocation logging is turned on, delivering prompts and responses to a single destination chosen based on access and retention needs: Amazon CloudWatch Logs for operational visibility or Amazon S3 for retention. AWS CloudTrail is turned on across agent accounts. Trace IDs are generated within each service, but correlation across asynchronous boundaries is partial, and sensitive data protection is inconsistent.

Level	Name	What it looks like
3	Defined	Decision artifacts and CloudTrail logs are delivered to a dedicated Amazon S3 bucket in a separate Log Archive account, aligned with the AWS Security Reference Architecture , with versioning turned on and bucket policies that deny delete and overwrite. CloudTrail log file validation is on. Amazon CloudWatch Logs data protection masks common sensitive patterns. AWS Distro for OpenTelemetry instruments agent code, correlation IDs are propagated through Amazon SQS and Amazon EventBridge , and AWS X-Ray produces end-to-end traces.

Level	Name	What it looks like
4	Proactive	Amazon Bedrock AgentCore Observability captures session, trace, and span hierarchies that include agent reasoning, tool calls, and memory operations. AgentCore Evaluations continuously scores correctness, helpfulness, tool selection accuracy, and safety, with alarms on score drift. Tamper-evident retention is achieved through bucket policies that deny delete and overwrite , MFA delete, and (where compliance requirements call for it) S3 Object Lock in governance mode after careful review of the operational implications. Amazon Athena with AWS Glue Data Catalog makes the artifact store queryable, and standard investigation queries are documented.

Level	Name	What it looks like
5	Optimized	Tiered retention moves artifacts from CloudWatch Logs through Amazon S3 Standard to Amazon Glacier with automated S3 Lifecycle transitions tied to data classification tags. AWS KMS customer-managed keys with automatic rotation protect all log destinations. Investigation-time queries return answers within a target investigation window defined for each customer based on data volume and incident-response objectives, anomaly detection runs on the artifact index, and the observability data feeds back into evaluation and guardrail tuning.

Common issues to watch for

- Only final outputs are captured, so incident responders can see what the agent did but not why it did it, which is the first question asked in nearly every investigation.
- Decision artifacts share the credentials and lifetime of the agent that generated them, which means an affected agent can modify or delete its own record of what happened.
- Trace context is re-generated at every asynchronous boundary without a complementary correlation identifier, breaking the chain exactly where investigations need it most.
- Sensitive data protection applied at inference time but not at logging time, so prompts and responses containing PII, secrets, or regulated fields end up unmasked in long-term storage. CloudWatch Logs has built-in data protection policies that mask known sensitive types in place. For S3 destinations, masking has to be applied at write time before the object lands in the

bucket, and any retroactive remediation requires re-processing existing objects. Plan the masking strategy per destination rather than assuming one control covers both.

- Artifacts are retained indefinitely without a queryable index, producing petabytes of S3 objects that are technically complete but practically unsearchable during a time-sensitive incident.

Best practices

- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)
- [AGENTSEC05-BP02 Implement distributed tracing for agent interactions](#)

AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage

Agent decisions are only auditable if the reasoning behind them is captured, preserved intact, and reachable at the speed investigations actually move. Tamper-evident artifact storage, attribution to the original trigger, and a queryable index turn raw log volume into forensic capability.

Desired outcome:

- You capture every agent decision, action, and reasoning step in tamper-evident, queryable storage, producing a complete and verifiable record of agent behavior.
- Each logged action includes attribution to the initiating source (a human user session, an upstream event, a schedule, or another agent), so logged actions can typically be traced back to what triggered them.
- You can reconstruct the full decision-making process for any agent action from stored artifacts, independent of the agent's own account of its reasoning.
- Cryptographic validation verifies log integrity, the agent's operational IAM role can't modify or delete its own decision history, and sensitive data (PII, secrets, regulated fields) is masked or redacted before logs are written to long-term storage.

Common anti-patterns:

- Logging only final agent outputs without intermediate reasoning, tool invocations, or decision points, making incident reconstruction impossible.
- Storing logs and decision artifacts in mutable storage without write-once protection, so logs can be deleted or modified after the fact.

- Storing decision artifacts in the same account and with the same permissions as the agent's operational resources, letting an affected agent modify or delete its own history.
- Retaining artifacts without a queryable index, so scanning raw S3 objects becomes impractical during time-sensitive investigations.

Benefits of establishing this best practice:

- Detailed logging of reasoning chains, tool invocations, and intermediate steps, stored independently from the agent's operational resources, supports reconstruction of agent behavior during investigations.
- Amazon S3 Object Lock and AWS CloudTrail log file validation provide cryptographic proof of log integrity for compliance and forensic purposes.
- Queryable artifact stores support retrospective behavioral analysis that surfaces patterns missed by real-time alerts.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Logging for agents has three hard requirements that ordinary application logging doesn't:

- Completeness (the reasoning chain, not just the outputs)
- Immutability (the agent whose behavior you are investigating can't be the entity that controls its own logs)
- Queryability at investigation speed (S3 scans are not fast enough when minutes matter)

Each of those requirements shapes a different piece of the architecture.

Start with the model invocation layer. Enable Amazon Bedrock model invocation logging to capture all inference requests and responses (input prompts, model outputs, token counts, latency metrics) with delivery to Amazon CloudWatch Logs for operational monitoring and Amazon S3 for long-term retention. Because agent prompts and model outputs often contain PII, secrets, or regulated data, apply data protection before content lands in long-term storage. [Amazon CloudWatch Logs data protection policies](#) automatically detect and mask sensitive types (credentials, personal identifiers, financial data) in log events. [Amazon Bedrock Guardrails](#) sensitive information filters anonymize or redact PII in input prompts and model responses at inference time, but that is

distinct from what the logs capture. Verify masking behavior across each destination in use (Amazon Bedrock Model Invocation Logs, AgentCore Observability, CloudWatch Logs, the S3 artifact store) and add write-time masking in agent code wherever the source doesn't mask on its own.

Decision artifacts require a separate trust boundary. Create a dedicated S3 bucket in a separate AWS account with versioning enabled, and use bucket policies that allow write from the agent account but deny delete and overwrite operations. That gives you an append-only artifact store the agent's operational IAM role can't tamper with. A consistent key schema (agent ID, session ID, timestamp, decision type) makes retrieval predictable during investigations. Capture the initiator on every decision. An agent can be invoked by:

- A human user session
- An Amazon EventBridge event
- An Amazon SQS message
- An Amazon CloudWatch alarm
- A scheduled rule
- Another agent

Log the identifiers that describe the trigger (IAM session or Amazon Cognito user for human requests, event source and event ID for events, alarm ARN for alarms, calling agent and session IDs for inter-agent calls) as structured fields so investigation queries can filter by trigger source.

For tamper-evidence on the bucket itself, default to bucket policies that deny delete and overwrite, MFA delete on the bucket, and versioning. Where compliance requirements call for stronger guarantees, consider enabling Amazon S3 Object Lock in governance mode, which allows users with specific IAM permissions to override retention settings when needed. Compliance mode helps prevent any user (including the root account) from deleting or shortening retention periods for the duration of the lock. Once configured, this mode is irreversible, and a misconfiguration of the retention period or scope can leave a customer unable to delete data they need to delete (for example, to meet right-to-be-forgotten requests). Use compliance mode only when there is a specific regulatory requirement for it, and validate the retention configuration against representative test data before applying it broadly.

[Amazon Bedrock AgentCore Observability](#) captures agent reasoning chains, tool invocations, and decision artifacts automatically for agents running on AgentCore Runtime. The session, trace, and span hierarchy records reasoning steps, tool calls with inputs and outputs, and memory operations.

AgentCore outputs span data for memory resources by default and publishes session-level metrics viewable on the Amazon CloudWatch generative AI observability page. For artifacts that need retention beyond the default observability window or specific compliance controls, write the full decision context to the dedicated S3 artifact store at each significant decision point. AWS Distro for OpenTelemetry (ADOT) extends coverage with custom metrics, logs, and spans in agent code.

[Amazon Bedrock AgentCore Evaluations](#) complements logging by continually scoring agent behavior on correctness, helpfulness, tool selection accuracy, and safety. Results publish to Amazon CloudWatch alongside observability data for a unified view, and Amazon CloudWatch alarms on evaluation scores detect behavioral drift outside acceptable thresholds.

Tamper-evidence at the audit-trail level uses AWS CloudTrail with log file validation across all accounts and regions where agents operate. Log file validation produces SHA-256 hashes and RSA signatures in a digest file that verifies log files have not been modified, deleted, or forged after delivery. A dedicated S3 bucket with cross-account access controls helps prevent the agent's own IAM role from modifying or deleting logs.

Amazon Athena with AWS Glue Data Catalog makes the decision artifact store queryable: an AWS Glue crawler scans the S3 artifact bucket and creates tables in the Data Catalog based on the artifact schema, and Athena runs SQL queries directly against S3 without loading data into a separate database. Investigation queries such as "find all decisions made by agent X that involved tool Y between dates A and B" become cheap to run. Document standard investigation queries for common security scenarios so investigators can work immediately during an incident. This pattern (logs to Amazon S3, cataloged by AWS Glue, queried with Amazon Athena) is an established forensic log analytics approach recommended in the AWS Well-Architected Security Pillar.

A lifecycle policy keeps storage costs proportional to access patterns. Hot logs live in Amazon CloudWatch Logs for operational monitoring (30 to 90 days), transition to Amazon S3 Standard for medium-term retention (1 to 2 years), and archive to Amazon Glacier for long-term compliance retention (7+ years). Tag objects with data classification and retention policy metadata so lifecycle transitions are automated.

Implementation steps

1. **Enable Amazon Bedrock model invocation logging:** Turn on Amazon Bedrock model invocation logging and deliver to both Amazon CloudWatch Logs and Amazon S3.
2. **Mask sensitive data before it lands in long-term storage:** Configure Amazon CloudWatch Logs data protection policies, verify masking behavior across every logging destination in use, and add write-time masking in agent code where the destination doesn't mask on its own.

3. **Create a dedicated, append-only artifact bucket in a Log Archive account:** Create an Amazon S3 bucket in a separate Log Archive account, aligned with the [AWS Security Reference Architecture](#), with versioning enabled and cross-account access controls that allow agent write access but deny delete and overwrite operations.
4. **Choose a retention-protection model:** Default to bucket-policy-based protection (deny delete, deny overwrite, MFA delete, versioning). Evaluate Amazon S3 Object Lock in governance mode where compliance requires stronger guarantees, and reserve compliance mode for cases where there is a specific regulatory requirement for it after validating the retention scope and duration against representative test data.
5. **Capture full traces through AgentCore Observability:** Use [Amazon Bedrock AgentCore Observability](#) to capture full agent execution traces, and for artifacts needing longer retention or compliance controls, write the full decision context to the dedicated S3 artifact store at each significant decision point.
6. **Record initiator attributes on every artifact:** Capture human session, event source and event ID, alarm ARN, schedule rule ARN, calling agent ID, and similar identifiers as structured fields on decision artifacts and log entries.
7. **Apply a consistent artifact key schema:** Use agentId, sessionId, timestamp, and decisionType as the key schema for efficient retrieval during investigations.
8. **Deploy AgentCore Evaluations with drift alarms:** Deploy [Amazon Bedrock AgentCore Evaluations](#) with built-in and custom evaluators, publish results to Amazon CloudWatch, and configure alarms on evaluation scores to detect behavioral drift.
9. **Enable CloudTrail log file validation:** Turn on AWS CloudTrail with log file validation across all accounts and regions, storing logs in a dedicated S3 bucket with cross-account access controls.
- 10 **Make artifacts queryable with Athena and AWS Glue:** Set up an AWS Glue crawler to scan the S3 artifact bucket and create tables in the Data Catalog, use Amazon Athena to query artifacts directly in S3, and document standard investigation queries for common security scenarios.
- 11 **Implement tiered retention with automation:** Define retention tiers (CloudWatch Logs for operational monitoring, S3 Standard for medium-term, Amazon Glacier for long-term compliance) with automated lifecycle transitions and data classification tagging.
- 12 **Encrypt all log and artifact storage:** Use customer-managed AWS KMS keys with key rotation enabled on every logging destination.

Resources

Related best practices:

- [AGENTSEC03-BP02 Separate agent and human user permission](#)
- [AGENTSEC04-BP02 Human-in-the-loop for critical decisions](#)
- [AGENTSEC05-BP02 Implement distributed tracing for agent interactions](#)
- [AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery](#)

Related documents:

- [Amazon Bedrock model invocation logging](#)
- [AgentCore Observability: Sessions, traces, and spans](#)
- [Build trustworthy AI agents with Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock AgentCore adds quality evaluations and policy controls](#)
- [AWS CloudTrail log file validation](#)
- [Amazon S3 Object Lock](#)
- [Amazon CloudWatch Logs data protection](#)
- [Amazon Athena documentation](#)
- [Querying AWS service logs with Amazon Athena](#)
- [Create a customizable cross-company log lake for compliance \(AWS Big Data Blog\)](#)
- [AWS Well-Architected Security Pillar SEC04-BP01: Configure service and application logging](#)
- [Considerations for addressing the core dimensions of responsible AI for Amazon Bedrock applications](#)
- [Security reference architecture for generative AI](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS CloudTrail](#)
- [Amazon CloudWatch](#)
- [Amazon S3](#)
- [Amazon Athena](#)
- [AWS Glue](#)
- [AWS Key Management Service](#)

AGENTSEC05-BP02 Implement distributed tracing for agent interactions

A request that hops across agents, queues, and event buses is only investigable if a single identifier follows it end-to-end. Tracing with both a trace ID for instrumented services and an application-level correlation ID for asynchronous boundaries makes cross-agent incidents reconstructable.

Desired outcome:

- You trace every request that flows through a multi-agent system end-to-end with a single correlation identifier, so security teams can reconstruct the complete chain of agent interactions for traced operations.
- Service maps give real-time visibility into agent dependencies and communication patterns.

Common anti-patterns:

- Generating new trace IDs at each agent boundary rather than propagating the original, breaking the correlation chain and making it impossible to link related actions.
- Tracing only synchronous agent interactions and omitting asynchronous operations (Amazon SQS messages, Amazon EventBridge events), creating gaps that obscure the full execution path.
- Not instrumenting tool invocations within agent traces, losing visibility into which external services were called and what data was exchanged during execution.

Benefits of establishing this best practice:

- Correlated traces across agent boundaries make every agent execution reconstructable after the fact.
- Service maps and trace analysis surface unexpected communication patterns, such as agents interacting with services outside their normal dependency graph.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Two identifiers do the work together. A *trace ID* is generated by the tracing system (Amazon CloudWatch Application Signals, AWS X-Ray, OpenTelemetry) and follows a request through

instrumented services. It is the identifier the tracing backend uses to reconstruct spans into a trace tree. A *correlation ID* is generated by the application and propagated end-to-end, and it survives boundaries where trace context is re-generated (most commonly asynchronous messaging channels, where the consumer frequently starts a new trace). Trace IDs give you the automatic correlation where instrumentation is continuous. Correlation IDs give you reliable linkage across the boundaries that instrumentation can't traverse.

[Amazon Bedrock AgentCore Observability](#) provides built-in tracing for agents deployed on AgentCore Runtime. Sessions represent complete user interactions, traces represent individual request-response cycles, and spans represent specific operations within a trace. AgentCore outputs span data for memory resources by default, and session-level metrics are viewable on the Amazon CloudWatch generative AI observability page. The built-in instrumentation captures the agent execution loop and propagates trace context across agent boundaries without custom code.

For deeper visibility or for agents not running on AgentCore Runtime, instrument agent code with AWS Distro for OpenTelemetry (ADOT) to generate traces compatible with AWS X-Ray and third-party observability platforms. Create spans for each significant operation within an agent (model invocations, tool calls, memory reads and writes, inter-agent communications) and configure X-Ray sampling rules to capture 100% of traces for security-critical operations while using statistical sampling for high-volume routine operations.

Correlation ID propagation is the primary concern in all agent-to-agent communications. Include both the correlation ID and the current trace ID in inter-agent messages, API calls, and event payloads, so the full execution chain can be reconstructed from any point. For asynchronous operations through Amazon SQS or Amazon EventBridge, propagate both IDs through message attributes. The correlation ID preserves end-to-end linkage even when the tracing system starts a new trace on the consumer side.

The Amazon CloudWatch generative AI observability page provides agent-specific session and trace metrics for AgentCore Runtime. For cross-service visualization that covers non-agent components in the same request path (databases, queues, downstream services), Amazon CloudWatch ServiceLens renders a service map of agent interactions and surfaces unexpected communication patterns. Amazon CloudWatch Logs Insights queries identify traces with unusual patterns: agents calling unexpected services, traces with abnormally high tool invocation counts, or traces that span unexpected geographic regions.

Implementation steps

1. **Verify built-in AgentCore tracing:** For agents on AgentCore Runtime, confirm [Amazon Bedrock AgentCore Observability](#) is capturing session, trace, and span data and review the default metrics on the Amazon CloudWatch generative AI observability page.
2. **Instrument custom agents with ADOT:** For custom agents or deeper visibility, instrument agent code with AWS Distro for OpenTelemetry to generate spans for model invocations, tool calls, memory operations, and inter-agent communications.
3. **Propagate correlation IDs through async boundaries:** Include the correlation ID and current trace ID in all inter-agent messages, API calls, and event payloads, and propagate them through Amazon SQS message attributes and Amazon EventBridge event detail for asynchronous operations.
4. **Configure X-Ray sampling by risk:** Capture 100% of traces for security-critical operations and statistical sampling for routine operations through AWS X-Ray sampling rules.
5. **Visualize service maps and detect anomalies:** Use Amazon CloudWatch ServiceLens to visualize agent service maps and build Amazon CloudWatch Logs Insights queries to detect anomalous trace patterns.
6. **Set trace retention:** Configure trace retention policies that match your incident investigation and compliance requirements.

Resources

Related best practices:

- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)
- [AGENTSEC06-BP04 Monitor and detect coordination anomalies](#)
- [AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery](#)
- [AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets](#)

Related documents:

- [AgentCore Observability: Sessions, traces, and spans](#)
- [Build trustworthy AI agents with Amazon Bedrock AgentCore Observability](#)
- [AWS X-Ray documentation](#)

- [AWS Distro for OpenTelemetry](#)
- [Amazon CloudWatch ServiceLens](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS X-Ray](#)
- [Amazon CloudWatch](#)

Secure multi-agent orchestration

Multi-agent systems introduce coordination challenges that don't exist in single-agent architectures. Agents need to discover peers, share capabilities, delegate tasks, and exchange context across trust boundaries. Without proper identity verification and communication security, agent impersonation and message tampering can affect entire multi-agent workflows. Standardized protocols like Agent-to-Agent (A2A) and Model Context Protocol (MCP) provide interoperability across frameworks, but organizations must layer security controls on top of these protocols to maintain trust boundaries.

AGENTSEC06: How do you secure multi-agent orchestration and coordination?

Capability intent

- Agents are segmented into trust zones based on role, capability, and risk profile, and inter-zone communication flows only along documented paths enforced at both the network and the application layer.
- Inter-agent messages that cross trust boundaries or traverse intermediary services are signed and encrypted at the message level, so payload integrity persists beyond the transport.
- Orchestration layers enforce schema validation, scoped IAM permissions, and circuit breakers, so a failure in one agent can't cascade through the workflow or divert it onto unexpected execution paths.
- Coordination patterns are monitored continually against established baselines, so deviations are detected proactively rather than discovered after the fact.

- Every coordination step is attributable to a verified agent identity and recorded for investigation. This includes agent card discovery (the [A2A protocol](#) mechanism by which one agent locates a peer and reads a JSON agent card describing its name, capabilities, supported skills, and authentication requirements), task delegation, and result collection.

Maturity levels

These levels summarize what each stage of maturity looks like for secure multi-agent orchestration as a whole.

Level	Name	What it looks like
1	Initial	Agents run in a flat network with no segmentation. Inter-agent messages rely on transport-level encryption alone, with no message signing or identity verification on the receiving side. Orchestration is unstructured, with broad IAM permissions on workflow APIs and no circuit breakers, and coordination patterns are not monitored.
2	Emerging	Basic network segmentation groups agents by trust tier, typically through separate security groups or Amazon VPCs . Amazon SQS server-side encryption helps protect queued messages with AWS KMS keys. AWS Step Functions orchestrates workflows with execution logging turned on, although

Level	Name	What it looks like
		IAM permissions remain broad and circuit breakers are not currently in place.
3	Defined	Message-level signing through AWS KMS asymmetric keys covers messages that cross trust boundaries or traverse queues and event buses, with separate keys per trust zone. Step Functions state machines are managed as code through AWS CloudFormation or the AWS Cloud Development Kit (AWS CDK) , input validation and circuit breakers are in place, and AWS PrivateLink keeps cross-zone traffic on private paths.

Level	Name	What it looks like
4	Proactive	Amazon Bedrock AgentCore Runtime with A2A protocol support structures inter-agent discovery and task delegation, and Cedar policies in AgentCore Policy enforce trust boundaries at the tool layer. Coordination metrics are captured as Amazon CloudWatch custom metrics with baselines , and CloudWatch anomaly detection flags deviations. Amazon GuardDuty findings are correlated with coordination logs in AWS Security Hub CSPM .

Level	Name	What it looks like
5	Optimized	Trust boundary configurations are continuously validated through AWS Config managed and custom rules , with alerts on any configuration that creates unauthorized cross-zone connectivity. AgentCore Evaluations feed tool selection accuracy and correctness scores into coordination monitoring as an early-warning layer. Incident response runbooks for coordination anomalies are exercised regularly, and validated patterns are folded back into the organization's reference architecture.

Common issues to watch for

- Multi-agent systems are deployed as a single trust zone, so an issue with any one agent can spread laterally across the entire network before the scope is understood.
- Teams rely on transport-level encryption for inter-agent traffic and treat message-level signing as unnecessary, which means messages sitting in queues or event buses can't be verified at consumption time.
- Orchestrator IAM permissions authorize any principal to start, stop, or modify any workflow in the account, rather than scoping access to the specific state machines each principal needs.
- Monitoring covers infrastructure metrics and individual agent health but ignores coordination metrics, so inter-agent message rate spikes, unexpected communication paths, and topology changes go undetected.

- [Amazon GuardDuty](#) findings and coordination logs are treated as separate data streams, leaving investigators without the context to tie an API anomaly to the specific multi-agent workflow it affected.

Best practices

- [AGENTSEC06-BP01 Encrypt and sign inter-agent messages](#)
- [AGENTSEC06-BP02 Implement workflow orchestration security controls](#)
- [AGENTSEC06-BP03 Establish trust boundaries between agents](#)
- [AGENTSEC06-BP04 Monitor and detect coordination anomalies](#)

AGENTSEC06-BP01 Encrypt and sign inter-agent messages

Transport-level encryption stops protecting the payload the moment a message lands in a queue or event bus. Message-level signing and encryption keyed per trust zone gives receiving agents cryptographic proof of sender identity and content integrity no matter how many hops the message made.

Desired outcome:

- You protect inter-agent messages at the message level with encryption and signing, independent of transport-level security.
- Receiving agents verify message signatures before acting on instructions from other agents.
- Messages stored in queues or event buses are encrypted at rest with keys scoped to the appropriate trust zone.
- Agents can't forge messages that appear to originate from other agents in the coordination workflow.

Common anti-patterns:

- Relying solely on transport-level encryption (TLS) without message-level signing, so messages stored in queues or event buses can't be verified for tampering at consumption time.
- Using a single shared encryption key across all agents, so one key exposure affects every inter-agent message rather than only those in one trust zone.
- Not verifying message signatures before acting on inter-agent instructions, letting an agent act on forged or tampered instructions without any integrity check.

Benefits of establishing this best practice:

- Message-level integrity verification persists beyond the transport layer, so messages stored in queues or event buses can be verified at consumption time.
- Scoped key management limits the impact of a key exposure to a single trust zone rather than the entire multi-agent system.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Two distinct properties are at stake: who sent the message (covered by AGENTSEC03-BP01 for authentication) and whether the content is what the sender actually sent. Transport-level encryption provides the second property only while the message is in transit. The moment the message lands in an Amazon SQS queue, Amazon EventBridge bus, or AWS Step Functions state machine, there is no transport to protect it, and any modification between delivery and consumption is invisible. Message-level signing provides cryptographic proof that a specific message was sent by a specific agent and has not been modified, regardless of how many intermediary services it passed through.

Scope matters. For direct agent-to-agent communication within a single trust zone, the transport-level security and authentication provided by [Amazon Bedrock AgentCore Runtime](#) (TLS 1.2+ with OAuth 2.0 or SigV4) is typically sufficient. Layer message signing on top when payload integrity needs to persist beyond the transport, or when a receiving agent needs cryptographic proof of which specific agent sent an instruction.

AWS KMS asymmetric keys are the mechanism. The sending agent signs the message payload through the KMS Sign API, and the receiving agent verifies the signature through the KMS Verify API before acting on the message. This gives end-to-end integrity verification that is independent of the transport layer: even if a message passes through multiple intermediary services (load balancers, queues, event buses), the signature proves it was created by the claimed sender and has not been modified. Separate keys per trust zone limit the scope affected by a single key exposure.

For asynchronous agent messaging through Amazon SQS, enable server-side encryption using AWS KMS customer-managed keys. Configure separate keys for different agent trust zones so a key exposure in one zone doesn't affect messages in other zones, and use SQS message attributes to carry the message signature alongside the encrypted payload so receiving agents verify both authenticity and integrity at consumption time.

AgentCore Runtime provides session isolation and built-in authentication for agents using the Agent-to-Agent (A2A) protocol. A2A handles authentication for inter-agent interactions, but for messages that cross trust boundaries, carry sensitive data, or pass through intermediary services, message-level signing layers on top of the protocol's built-in protections.

AWS PrivateLink routes inter-agent communications through private network endpoints, keeping traffic off the public internet. That complements message-level encryption by reducing the network exposure of inter-agent traffic. Store signing key ARNs in Parameter Store, a capability of AWS Systems Manager, configure automatic key rotation in AWS KMS with a rotation period matching your security requirements, and configure Amazon CloudWatch alarms for key usage anomalies (unexpected signing operations from agents that should only be verifying, signing volume spikes suggesting a runaway loop).

Implementation steps

1. **Create trust-zone-scoped KMS key pairs:** Provision AWS KMS asymmetric key pairs for message signing, with separate keys for each agent trust zone.
2. **Sign messages on send:** Implement message signing in sending agents through the AWS KMS Sign API, attaching the signature as a message metadata attribute.
3. **Verify signatures on receive:** Implement signature verification in receiving agents through the AWS KMS Verify API, rejecting any message that fails verification before processing.
4. **Encrypt SQS queues with customer-managed keys:** Enable server-side encryption on Amazon SQS queues used for asynchronous inter-agent messaging, using customer-managed AWS KMS keys scoped per trust zone.
5. **Layer signing on top of A2A for cross-boundary traffic:** For agents on AgentCore Runtime using A2A, add message-level signing for communications that cross trust boundaries on top of the protocol's built-in authentication.
6. **Route through PrivateLink:** Configure AWS PrivateLink for inter-agent service communications to keep traffic on private network endpoints.
7. **Manage keys and alert on anomalies:** Store signing key ARNs in Parameter Store, a capability of AWS Systems Manager, configure automatic key rotation in AWS KMS, and set Amazon CloudWatch alarms for key usage anomalies.

Resources

Related best practices:

- [AGENTSEC03-BP01 Implement strong authentication for agent identities](#)
- [AGENTSEC06-BP02 Implement workflow orchestration security controls](#)
- [AGENTSEC06-BP03 Establish trust boundaries between agents](#)

Related documents:

- [Introducing agent-to-agent protocol support in Amazon Bedrock AgentCore Runtime](#)
- [Digital signing with the new asymmetric keys feature of AWS KMS](#)
- [Code signing using ACM Private CA and AWS KMS asymmetric keys](#)
- [AWS KMS documentation](#)
- [Amazon SQS security best practices](#)

Related services:

- [AWS Key Management Service](#)
- [Amazon SQS](#)
- [AWS Systems Manager](#)
- [AWS PrivateLink](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTSEC06-BP02 Implement workflow orchestration security controls

The orchestration layer is where a single weak check cascades into a system-wide failure. Access controls on state machines, input validation on transitions, and circuit breakers on agent tasks keep multi-agent workflows on approved execution paths instead of unexpected ones.

Desired outcome:

- The workflow orchestration layer enforces strict access controls that help prevent unauthorized modification of workflow definitions or execution state.
- State machine validation helps keep workflows on expected execution patterns, and circuit breakers are designed to stop failures in one agent from cascading through the entire workflow.

- You log all workflow executions with enough detail to reconstruct the execution path for security investigations.

Common anti-patterns:

- Granting broad IAM permissions to start, stop, or modify Step Functions workflows without restricting access to specific state machines, letting any principal with workflow permissions modify or trigger any workflow in the account.
- Not implementing input validation in state machine definitions, letting crafted input payloads direct workflows into unexpected execution paths.
- Failing to implement circuit breakers, so a single failing agent cascades failures through the entire workflow with no automatic mechanism to stop retrying a broken step.
- Using overly permissive retry configurations, letting an agent repeatedly attempt the same operation before any circuit breaker triggers and potentially amplifying the original issue.

Benefits of establishing this best practice:

- State validation and input schema enforcement keep workflows within defined boundaries.
- Circuit breakers automatically stop cascading failures and route affected executions to quarantine paths for investigation.
- AWS Step Functions logging captures every state transition, input, output, and error event for full execution reconstructability.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Two orchestration patterns are in scope here. AWS Step Functions state machines handle deterministic workflows where the execution path is defined in JSON and the orchestrator enforces sequencing. [Amazon Bedrock AgentCore Runtime](#) with the A2A protocol handles agent-delegated workflows where an orchestrator agent dynamically decides which sub-agents to invoke. Most of the controls below apply to both, and the differences are called out inline.

Start with IAM. Configure policies for AWS Step Functions that restrict workflow start, stop, and modification permissions to specific principals and state machines, use resource-based policies on state machine definitions to help prevent unauthorized modification, and implement IAM

Conditions that restrict execution to approved input schemas. Manage state machine definitions as infrastructure as code through AWS CloudFormation or AWS CDK, which helps prevent informal modifications and provides version-controlled change history.

Input validation belongs inside the state machine definition, not outside it. With Step Functions' built-in JSONPath filtering and AWS Lambda validation functions, you can validate that workflow inputs conform to expected schemas before passing them to agent tasks, rejecting inputs that deviate from expected patterns. Step Functions' error handling catches and logs validation failures without exposing error details to callers.

Circuit breakers use Step Functions' error handling and retry logic. Set conservative retry limits with exponential backoff for agent task failures, and implement catch states that route failed executions to a quarantine path rather than retrying indefinitely. Amazon EventBridge emits circuit breaker events when failure thresholds are exceeded, triggering alerts and automated remediation. For multi-agent workflows using the A2A protocol on AgentCore Runtime, the structured request lifecycle (agent card discovery, task delegation, result collection) provides natural points to validate inputs, check authorization, and apply circuit breaker logic before proceeding.

Execution logging makes the orchestration auditable. Enable Step Functions execution logging to Amazon CloudWatch Logs at the ALL level to capture all state transitions, input/output data, and error events. Configure log retention policies aligned with compliance requirements and create Amazon CloudWatch Logs Insights queries for common investigation scenarios such as identifying workflows that took unexpected execution paths or triggered circuit breakers.

Implementation steps

1. **Scope Step Functions IAM to specific state machines:** Configure IAM policies with least-privilege access scoped to specific state machines and execution operations.
2. **Manage state machines as IaC:** Use AWS CloudFormation or AWS CDK for state machine definitions to help prevent unauthorized modifications and keep version history.
3. **Validate inputs inside state machines:** Implement input validation in state machine definitions using JSONPath filtering and AWS Lambda validation functions, rejecting inputs that deviate from expected schemas.
4. **Configure circuit breakers with catch states:** Set conservative retry limits and implement catch states that route failures to quarantine paths rather than retrying indefinitely.
5. **Log every execution at the ALL level:** Enable Step Functions execution logging to Amazon CloudWatch Logs at the ALL level with retention policies aligned to compliance requirements.

6. **Alarm on circuit breaker events:** Create Amazon CloudWatch alarms for circuit breaker triggers and Amazon EventBridge rules to route workflow security events to the monitoring pipeline.

Resources

Related best practices:

- [AGENTSEC04-BP02 Human-in-the-loop for critical decisions](#)
- [AGENTSEC06-BP01 Encrypt and sign inter-agent messages](#)
- [AGENTSEC06-BP03 Establish trust boundaries between agents](#)
- [AGENTREL07-BP01 Design workflows in stages with incremental recovery](#)
- [AGENTREL07-BP02 Enable automatic recovery from agent execution failures](#)

Related documents:

- [AWS Step Functions security documentation](#)
- [Step Functions error handling](#)
- [Using the circuit breaker pattern with Step Functions and DynamoDB](#)
- [AWS Prescriptive Guidance: Circuit breaker pattern](#)
- [Introducing agent-to-agent protocol support in Amazon Bedrock AgentCore Runtime](#)

Related services:

- [AWS Step Functions](#)
- [AWS Identity and Access Management](#)
- [Amazon CloudWatch](#)
- [Amazon EventBridge](#)
- [AWS CloudFormation](#)
- [Amazon Bedrock AgentCore](#)

AGENTSEC06-BP03 Establish trust boundaries between agents

A flat agent network gives every affected agent a direct path to every other one. Trust zones segmented at the network and IAM layers, with application-layer verification of caller identity, stop one affected agent from escalating across the whole system.

Desired outcome:

- Agents operate within clearly defined trust zones, accepting instructions only from authorized coordinators and rejecting requests from agents outside their trust boundary.
- Network segmentation enforces trust boundaries at the infrastructure layer and IAM policies enforce them at the API layer.
- An affected agent in one trust zone can't directly issue instructions to agents in higher-trust zones without passing through authorization controls.

Common anti-patterns:

- Deploying all agents in a flat network without segmentation, letting any agent communicate directly with any other regardless of trust level so an issue spreads laterally.
- Relying on network-level trust boundaries alone without application-layer authorization, so any agent that reaches another agent's endpoint can issue instructions.
- Not validating the identity of the coordinator agent before executing instructions, letting any agent impersonate a coordinator and issue unauthorized commands.
- Treating all internal agents as implicitly trusted while implementing trust boundaries only for external-facing agents, producing a flat internal trust model that amplifies the impact of any internal issue.

Benefits of establishing this best practice:

- Trust zone segmentation contains the impact of an affected agent to its own trust zone, helping prevent lateral movement.
- Layered enforcement at both the network level (VPC segmentation, security groups) and the application level (IAM policies, agent identity validation) provides defense-in-depth.
- Documented trust architecture supports automated compliance checks that catch drift as configurations evolve.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Trust boundary controls apply regardless of the inter-agent protocol used, whether A2A, MCP, or custom REST. The network-layer controls (VPC segmentation, security groups, AWS PrivateLink) and IAM-layer controls (resource-based policies, IAM Conditions) enforce boundaries independent of the application protocol. Protocol-specific guidance applies on top of these common controls.

A trust zone architecture starts with tiers that reflect actual risk: public, internal operational, privileged. Enforce the tiers at the network with separate Amazon VPCs or VPC security groups, and use Amazon VPC peering or AWS Transit Gateway with route table controls to restrict inter-zone communication to only the required paths. Network segmentation alone doesn't verify the caller's identity, so pair it with application-layer authorization.

[Amazon Bedrock AgentCore Runtime](#) A2A protocol support provides a structured framework for inter-agent communication with built-in session isolation and authentication. When agents discover peers through A2A agent cards, the card schema advertises the agent's capabilities and authentication requirements. Configure agents to accept A2A connections only from coordinators whose agent cards match the expected identity and trust level. For agents not using A2A, Amazon API Gateway with AWS Lambda authorizers implements custom agent-to-agent authorization logic that validates agent identity tokens and enforces trust level requirements.

Resource-based policies on agent endpoints explicitly list the IAM principals authorized to invoke each agent. IAM Conditions restrict invocations to agents within the same trust zone or to specific coordinator agent roles. AWS PrivateLink keeps cross-zone agent communications on private network paths. [Policy in Amazon Bedrock AgentCore](#) reinforces trust boundaries at the tool layer: Cedar policies can include conditions on the calling principal's identity and trust level, so even if an agent can reach another agent's tools through the gateway, the policy engine blocks tool calls that violate trust zone rules.

Compliance validation detects drift from the intended network posture. AWS Config managed rules, `vpc-sg-open-only-to-authorized-ports` for unintended public ingress, `restricted-ssh` for SSH access from `0.0.0.0/0`, `vpc-sg-port-restriction-check` for port-level restrictions, cover baseline network hygiene. Trust-zone-specific validation (that security group rules reference only CIDR ranges or security group IDs from the same trust zone) needs custom AWS Config rules backed by AWS Lambda, and alarms fire on any configuration change that would create unauthorized cross-zone connectivity.

Implementation steps

1. **Design trust zone tiers:** Define tiers (public, internal operational, privileged) and document the authorized communication paths between zones.
2. **Segment at the network layer:** Create separate Amazon VPCs or security groups for each trust zone and configure network controls (VPC peering, AWS Transit Gateway route tables) to enforce zone boundaries.
3. **Enforce identity at the application layer:** For agents on [Amazon Bedrock AgentCore Runtime](#), configure A2A agent card discovery with authentication requirements that enforce trust-level validation. For agents not on AgentCore Runtime, use Amazon API Gateway with AWS Lambda authorizers for custom trust boundary enforcement.
4. **Apply resource-based IAM policies:** List only authorized coordinator principals in each agent endpoint's resource policy, with IAM Conditions restricting invocations by trust zone.
5. **Reinforce at the tool layer with Policy:** Configure Cedar policies in [Policy in Amazon Bedrock AgentCore](#) with conditions on calling principal identity and trust level.
6. **Keep cross-zone traffic private:** Implement AWS PrivateLink for cross-zone agent communications.
7. **Validate configurations continually:** Deploy AWS Config managed rules (vpc-sg-open-only-to-authorized-ports, restricted-ssh, vpc-sg-port-restriction-check) for baseline hygiene and custom AWS Config rules for trust-zone-specific validation, alarming on any change that would create unauthorized cross-zone connectivity.

Resources

Related best practices:

- [AGENTSEC03-BP03 Implement least privilege with dynamic boundaries](#)
- [AGENTSEC06-BP01 Encrypt and sign inter-agent messages](#)
- [AGENTSEC06-BP02 Implement workflow orchestration security controls](#)

Related documents:

- [Introducing agent-to-agent protocol support in Amazon Bedrock AgentCore Runtime](#)
- [Secure AI agents with Policy in Amazon Bedrock AgentCore](#)
- [AWS VPC security best practices](#)

- [AWS Config managed rules reference](#)
- [AWS Config custom rules with Lambda](#)
- [Security reference architecture for generative AI](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Identity and Access Management](#)
- [Amazon VPC](#)
- [Amazon API Gateway](#)
- [AWS PrivateLink](#)
- [AWS Config](#)

AGENTSEC06-BP04 Monitor and detect coordination anomalies

Distributed tracing tells you what happened on one request. Coordination monitoring tells you when many requests start behaving differently from the baseline. Tracking inter-agent message rates, workflow frequencies, and topology changes against established baselines catches issues through their observable impact on coordination before they escalate into security events.

Desired outcome:

- You detect anomalous coordination patterns such as unexpected agent communication paths, unusual interaction frequencies, or coordination latency spikes in near real time and trigger alerts for investigation.
- You establish baseline coordination profiles for each agent workflow, so statistical anomaly detection catches deviations before they cause significant impact.

Common anti-patterns:

- Monitoring only infrastructure metrics (CPU, memory, and network) without tracking agent-specific coordination metrics, missing the coordination-level signals most indicative of multi-agent issues.
- Not establishing coordination baselines before deploying anomaly detection, which produces either excessive false positives or missed detections.

- Treating Amazon GuardDuty findings and agent coordination logs as separate data streams, leaving investigators without the multi-agent context that turns API-level anomalies into useful signal.

Benefits of establishing this best practice:

- Behavioral baselines catch coordination deviations before they propagate across the multi-agent system.
- Service maps compare observed communication paths against the expected trust boundary architecture, validating topology continually.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Distributed tracing (AGENTSEC05-BP02) reconstructs what happened during a specific request. Coordination anomaly detection is a different problem. It is a proactive early-warning system that watches whether coordination patterns across many requests over time are drifting from the baseline. Tracing is reactive and investigation-focused. Coordination monitoring is preventive and baseline-focused.

Start by defining coordination metrics for each multi-agent workflow:

- Inter-agent message rates
- Workflow execution frequencies
- Agent response latencies
- Error rates per agent pair
- Coordination graph topology changes

Publish these as Amazon CloudWatch custom metrics, establish baselines by collecting them during normal operation, and configure Amazon CloudWatch anomaly detection to automatically identify statistical deviations.

[Amazon Bedrock AgentCore Observability](#) provides the foundation. The session, trace, and span hierarchy captures inter-agent interactions, and the default metrics on the Amazon CloudWatch generative AI observability page surface session-level patterns. For agents using the A2A protocol on AgentCore Runtime, the structured request lifecycle (agent card discovery, task delegation,

result collection) generates observable events at each coordination step, which you can use to build coordination topology maps and detect when agents communicate outside their expected patterns.

[Amazon Bedrock AgentCore Evaluations](#) complements coordination monitoring by continually scoring agent behavior on tool selection accuracy, correctness, and other quality dimensions. If a coordinating agent starts selecting unexpected tools or producing incorrect outputs, evaluation-score drops can serve as an early warning signal before the coordination anomaly becomes visible at the workflow level. Amazon CloudWatch alarms on evaluation scores layered with coordination metrics give you a two-stage detection approach.

Amazon GuardDuty monitors API call patterns for all agent IAM roles. Its machine learning models detect unusual call patterns: an agent suddenly calling services it has never accessed before, or calling APIs at unusual times or from unexpected locations. Integrate GuardDuty findings with AWS Security Hub CSPM for centralized prioritization, and correlate them with agent coordination logs to connect API-level anomalies to specific multi-agent workflows. Amazon CloudWatch Logs Insights queries add another layer, analyzing agent coordination logs for patterns such as agents receiving instructions from unexpected sources, coordination loops that may indicate runaway behavior, and agents attempting to access resources outside their defined scope. Schedule these queries to run periodically and publish results to a security dashboard.

Implementation steps

1. **Define and publish coordination metrics:** Capture inter-agent message rates, execution frequencies, response latencies, and error rates per agent pair through Amazon CloudWatch custom metrics for each multi-agent workflow.
2. **Establish baselines and enable anomaly detection:** Collect metrics during normal operation and configure Amazon CloudWatch anomaly detection on key metrics.
3. **Build topology maps:** Use [Amazon Bedrock AgentCore Observability](#) service maps and A2A request lifecycle events to build coordination topology maps, and alert on unexpected topology changes that deviate from the documented trust boundary architecture.
4. **Layer evaluations as early warning:** Deploy [Amazon Bedrock AgentCore Evaluations](#) to continually score agent behavior, and configure Amazon CloudWatch alarms on evaluation scores as an early-warning layer for coordination issues.
5. **Correlate GuardDuty with coordination logs:** Enable Amazon GuardDuty for all agent accounts, integrate findings with AWS Security Hub CSPM, and create correlation rules that connect API anomalies to specific agent coordination logs.

6. **Run Logs Insights queries on a schedule:** Build Amazon CloudWatch Logs Insights queries for coordination security event patterns and publish results to a security dashboard on a scheduled cadence.
7. **Document the response runbook:** Establish an incident response runbook for coordination anomaly alerts that defines investigation steps, escalation paths, and remediation actions.

Resources

Related best practices:

- [AGENTSEC06-BP01 Encrypt and sign inter-agent messages](#)
- [AGENTSEC06-BP03 Establish trust boundaries between agents](#)
- [AGENTSEC07-BP04 Behavioral anomaly detection and agent containment](#)
- [AGENTREL02-BP03 Implement behavioral anomaly detection and monitoring](#)

Related documents:

- [Build trustworthy AI agents with Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock AgentCore adds quality evaluations and policy controls](#)
- [Amazon GuardDuty documentation](#)
- [Amazon CloudWatch anomaly detection](#)

Related services:

- [Amazon CloudWatch](#)
- [Amazon GuardDuty](#)
- [Amazon Bedrock AgentCore](#)
- [AWS Security Hub CSPM](#)

Human oversight protection and agent containment

Human oversight is a key control in agentic systems, but it is only effective if reviewers can make informed decisions without being overwhelmed or misled. At the same time, you must detect and contain agents that deviate from their intended behavior before they can affect other parts

of the system. This capability covers both sides: protecting the quality of human review, and implementing detection and containment mechanisms for agents that operate outside their defined boundaries.

AGENTSEC07: How do you protect human oversight from manipulation and detect rogue agents?

Capability intent

- Review queues are prioritized and load-balanced, and reviewer workload is monitored for signs of rubber-stamping, so human reviewers make informed decisions rather than approving under pressure.
- Each decision presented for review carries calibrated context, including confidence scores, policy-check results, anomaly flags, and similar past decisions, so reviewers match their scrutiny to the actual risk of the action.
- High-risk agent actions receive independent review from multiple qualified reviewers, with blind review, consensus logic, and escalation for disagreement.
- Behavioral baselines exist for each agent, and deviations trigger rapid containment through credential revocation and circuit-breaker activation, with forensic state captured before isolation.
- Security assessments combine continuous automated scanning with scheduled red team and purple team exercises targeted at agent-specific manipulation scenarios, and findings flow back into guardrails, detection rules, and incident response runbooks. Red team exercises are adversarial simulations where a dedicated team attempts to compromise the agent system using the techniques a real attacker would use. Purple team exercises run the same scenarios collaboratively with the defenders so detection rules and runbooks are updated based on what was observed during the simulation.

Maturity levels

These levels summarize what each stage of maturity looks like for human oversight protection and agent containment as a whole.

Level	Name	What it looks like
1	Initial	Review pipelines are informal and routed to whoever is available, with no workload metrics or priority classification. Reviewers evaluate agent decisions without confidence scores, historical context, or independent verification. Agent behavior is observed only through infrastructure metrics, and containment depends on manual investigation after the fact. Security assessments are one-off or absent.
2	Emerging	Review requests flow through a documented queue with coarse priority tiers, and single-reviewer approval applies to most agent decisions. Agent outputs include generic grounding checks, but reviewers don't see historical patterns or manipulation warnings. Generic security assessments cover agent deployments, and agent-specific red team scenarios are limited.
3	Defined	A prioritized review pipeline backed by Amazon SQS and Amazon DynamoDB tracks time-to-decision

Level	Name	What it looks like
		and approval-rate metrics, with alarms on rubber-stamping patterns. Reviewers see grounding scores from Amazon Bedrock Guardrails contextual grounding checks, deviation flags against historical decisions, and plain-language warnings for flagged manipulation attempts. High-risk operations require independent review from more than one person using AWS Step Functions parallel states . Behavioral metrics are published as custom metrics with Amazon CloudWatch anomaly detection , and findings from Amazon GuardDuty and Amazon Macie are centralized in AWS Security Hub CSPM . Red team exercises run on a defined cadence using the OWASP Top 10 for Agentic Applications as the starting scenario library.

Level	Name	What it looks like
4	Proactive	Review quality is managed through dashboards and composite alarms that automatically escalate or quarantine when review-behavior metrics degrade. Blind review and consensus logic are the default for high-risk operations, with disagreements routed to senior reviewers. Containment is automated through Amazon EventBridge rules that capture forensic state to Amazon S3, revoke credentials by attaching a deny-all policy to the agent's IAM role, and broadcast quarantine events for circuit breakers in dependent workflows. Purple team activities run after each assessment cycle and update Amazon Bedrock Guardrails configurations, detection rules, and runbooks based on observed patterns.

Level	Name	What it looks like
5	Optimized	Review quality metrics, agent behavioral baselines, and containment procedures are continuously tuned based on outcomes, and new attack techniques feed directly into detection and response tooling. Containment runbooks are regularly exercised through AWS Fault Injection Service to validate that isolation, circuit breakers, and forensic capture behave as expected. Purple team findings, Amazon Bedrock AgentCore evaluation trends, and runtime threat signals drive continuous refinement of guardrails, detection thresholds, and reviewer training.

Common issues to watch for

- Review capacity is planned by volume alone, without metrics that detect rubber-stamping when queues back up, so human oversight degrades silently under load as reviewers approve faster than they can evaluate.
- Reviewers see agent recommendations without confidence scores, anomaly flags, or similar past decisions, and apply the same level of scrutiny to high-risk and low-risk actions because they have no basis for calibrating their attention.
- High-risk operations depend on a single approver, or multiple reviewers are allowed to see each other's decisions before submitting their own, turning the review process into either a single point of failure or an exercise in anchoring bias.

- Rogue-agent detection relies on infrastructure metrics rather than per-agent behavioral baselines, and quarantine requires a human to act, so an agent operating outside its intended scope keeps running for hours as responders mobilize.
- Security assessments are one-time, generic, or disconnected from the agents they are meant to protect, so findings never translate into updated guardrails, detection rules, or incident runbooks.

Best practices

- [AGENTSEC07-BP01 Implement cognitive load management](#)
- [AGENTSEC07-BP02 Clear confidence indicators and manipulation warnings](#)
- [AGENTSEC07-BP03 Multiple reviewers for critical operations](#)
- [AGENTSEC07-BP04 Behavioral anomaly detection and agent containment](#)
- [AGENTSEC07-BP05 Regular security assessments and red teaming](#)

AGENTSEC07-BP01 Implement cognitive load management

A human reviewer is only as effective as the workload lets them be. Prioritization, queue management, and maximum review rates keep human oversight grounded in genuine judgment rather than fatigue-driven rubber-stamping.

Desired outcome:

- Human reviewers receive a manageable volume of well-prioritized decisions, with sufficient context and time to make informed judgments.
- You monitor review queues for backlog accumulation, with automatic escalation or load balancing helping prevent any single reviewer from being overwhelmed.
- Review quality metrics detect signs of rubber-stamping that indicate cognitive overload.

Common anti-patterns:

- Routing all agent decisions requiring review to a single queue without prioritization, so high-priority security decisions wait behind routine approvals.
- Not monitoring reviewer workload or queue depth, letting backlogs accumulate silently until reviewers begin approving without adequate evaluation.

- Setting no maximum review rate per person, so a single reviewer can be assigned an unlimited number of decisions in a short period.

Benefits of establishing this best practice:

- Workload management keeps reviewers in a position to make genuine, informed decisions rather than rubber-stamping under pressure.
- Review quality metrics (average review time, approval rate) surface when the oversight process is breaking down, enabling intervention before it fails silently.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Amazon SQS standard queues don't support priority ordering natively, so the prioritization layer has to sit above the queue. A coarse-grained option is separate Amazon SQS queues per priority tier (high, medium, low), where the assignment function polls high-priority first and falls back to lower tiers only when the upper one is empty. A more flexible option uses a single ingest queue consumed by an AWS Lambda function that classifies each request by priority and writes it to an Amazon DynamoDB table with a sort key based on priority and submission time. The reviewer assignment function queries DynamoDB for the highest-priority unassigned items, marks them as assigned, and delivers them to the reviewer. The DynamoDB pattern gives you full control over prioritization logic, supports re-prioritization of pending items, and keeps a durable record of all review requests regardless of their current state. Items that are not immediately assigned stay in DynamoDB rather than sitting in a queue with an expiring visibility timeout.

Priority classification is about potential impact of the action proceeding incorrectly: the more damaging a mistaken approval would be, the higher the priority. Concrete factors include data sensitivity (PII, financial records, healthcare information), reversibility (deletes, external communications, and financial transactions can't be undone), whether the action is a first-time operation for this agent (no behavioral baseline, so reviewers can't pattern-match to shortcut judgment, and the first-time operation is itself a signal worth investigating), and time sensitivity (operations that become harder to reverse over time, or where delay has its own cost). Automate the classification by tagging review requests with metadata from the agent's tool invocation context, data classification tags on target resources, and the agent's historical usage patterns.

Amazon CloudWatch watches the DynamoDB review table for backlog accumulation (unassigned items by priority tier), average time-to-assignment, and average time-to-decision. Alarms fire on high-priority items remaining unassigned beyond defined thresholds.

Reviewer load balancing distributes decisions across available reviewers based on current workload. Amazon DynamoDB tracks reviewer assignment counts and availability, and an AWS Lambda-based assignment function routes new decisions to the reviewer with the lowest current load. Configure maximum assignment limits per reviewer per time window to help prevent overload.

Review quality metrics are the trailing indicator. Track average review time, approval rate, and decision reversal rate (cases where a second reviewer overrides the first), publish the metrics to Amazon CloudWatch, and alarm on patterns that suggest rubber-stamping: unusually short review times or abnormally high approval rates during periods of high queue volume. Automatic escalation routes unreviewed decisions past defined time thresholds to senior reviewers, or triggers a safe default (typically blocking the operation) to help prevent indefinite delays.

Approval bounds, how long an approval remains valid, whether it can be revoked, whether high-risk operations require step-up re-confirmation, are covered in AGENTSEC04-BP02, which details the persistent-trust patterns that determine the scope and lifetime of each approval decision.

Implementation steps

1. **Set up the ingest-classify-store pipeline:** Use an Amazon SQS ingest queue, an AWS Lambda classifier that assigns priority, and an Amazon DynamoDB review table with a sort key on priority and submission time.
2. **Build the reviewer assignment function:** Query the DynamoDB table for the highest-priority unassigned items, mark them as assigned, and deliver them to the appropriate reviewer.
3. **Cap assignments per reviewer and escalate:** Set maximum review assignment limits per reviewer per time window and configure automatic escalation when limits are reached or high-priority items age beyond thresholds.
4. **Measure review quality:** Track average review time, approval rate, and reversal rate in Amazon CloudWatch, and configure alarms on patterns that suggest rubber-stamping.
5. **Monitor the review table:** Alarm on backlog accumulation by priority tier, average time-to-assignment, and average time-to-decision, and alert when high-priority items age beyond thresholds.
6. **Review load metrics periodically:** Use cognitive load metrics to refine prioritization logic and reviewer capacity planning on a regular cadence.

Resources

Related best practices:

- [AGENTSEC04-BP02 Human-in-the-loop for critical decisions](#)
- [AGENTSEC07-BP03 Multiple reviewers for critical operations](#)
- [AGENTSEC07-BP04 Behavioral anomaly detection and agent containment](#)

Related documents:

- [Amazon SQS documentation](#)
- [AWS Step Functions human approval](#)
- [Implement human-in-the-loop confirmation with Amazon Bedrock Agents](#)

Related services:

- [Amazon SQS](#)
- [Amazon DynamoDB](#)
- [AWS Lambda](#)
- [Amazon CloudWatch](#)
- [Amazon SNS](#)

AGENTSEC07-BP02 Clear confidence indicators and manipulation warnings

Workload-managed reviewers still make poor decisions when they lack the context to evaluate what the agent is recommending. Surfacing agent confidence, manipulation flags, and historical comparisons lets reviewers calibrate scrutiny to the actual risk of each decision.

Desired outcome:

- Human reviewers see agent confidence scores, uncertainty indicators, and manipulation warning flags alongside each decision, letting them calibrate scrutiny appropriately.
- Historical context and similar past decisions are surfaced so reviewers can identify when an agent is recommending an action that deviates from established patterns.

Common anti-patterns:

- Presenting agent decisions without confidence scores or uncertainty indicators, leaving reviewers unable to distinguish high-confidence recommendations from speculative outputs.
- Not surfacing historical context or similar past decisions, so every recommendation looks equally plausible without a baseline of "what normally happens here."
- Displaying confidence scores without explaining their meaning or limitations, leading reviewers to over-trust high-confidence outputs without appropriate skepticism.

Benefits of establishing this best practice:

- Confidence scores and historical context help reviewers calibrate scrutiny to the actual risk of each decision.
- Deviation flags draw reviewer attention to the decisions that most need it when an agent's recommendation differs from historical patterns.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Start with what the reviewer is being asked to decide. Two patterns are common and require different signals. In the **evaluator pattern**, the agent recommends an action and the reviewer decides whether the recommendation is correct. The useful signal is the agent's confidence in its own output (a low score suggests the recommendation may be wrong). In the **gate pattern**, the agent wants to perform a high-risk action and the reviewer is a policy gate deciding whether the action should be allowed. The agent's own confidence is less useful because the agent would not have proposed the action if it thought it was wrong. For gate-pattern reviews, the useful signal comes from systems independent of the agent: anomaly detection, policy checks, and manipulation-warning flags from the input-validation pipeline.

For evaluator-pattern reviews, configure [Amazon Bedrock Guardrails contextual grounding](#) to generate a grounding score that checks whether the response is supported by the source material the agent was given. Surface that score alongside review notifications with plain-language explanations (for example, "the agent's response isn't well supported by the source material, and independent verification is recommended") so reviewers know what the score means rather than guessing. For gate-pattern reviews, draw from checks independent of the agent: anomaly detection (AGENTSEC07-BP04) evaluates whether the action deviates from baseline behavior,

and manipulation-warning flags raised by the input-validation pipeline (AGENTSEC04-BP01 and AGENTSEC08-BP01) signal when the request itself looks adversarial. [Amazon Bedrock Guardrails automated reasoning](#) provides a complementary deterministic check that validates agent outputs against a formally authored policy (for example, "users in region X can't perform operation Y"). It is most useful as context on the reviewer's screen ("this passed policy check X at Y time") rather than as the primary decision signal, because actions that pass automated reasoning typically don't need human gating unless the policy itself is known to have gaps or the action is irreversible enough to warrant redundant human sign-off. Surface each signal with the same plain-language framing so the reviewer knows what each number or flag means.

Historical context makes anomalies visible. Store decisions and their confidence scores in Amazon DynamoDB for fast retrieval during review. When a new decision comes up, query DynamoDB for similar past decisions (same operation type, same agent, similar parameters) and surface them alongside the current request. Flag the current decision if its confidence score deviates significantly from the historical average for that operation type.

[Amazon Bedrock AgentCore Evaluations](#) adds the quality-trend signal. Built-in evaluators cover correctness (does the output match expected answers on a test set), helpfulness, tool-selection accuracy (does the agent pick the right tool for the task), and safety. Add custom evaluators for domain-specific criteria: policy adherence, format conformance (does the output match the schema downstream systems expect), and goal attainment (did the agent actually accomplish what the user asked). Prioritize evaluators that measure things a reviewer could not easily verify themselves in the time they have during a review. If the reviewer can check it in ten seconds, it isn't what the evaluator is for. An agent whose evaluation scores are trending downward is a signal the reviewer needs to see alongside the specific decision in front of them.

Amazon Quick dashboards visualize decision patterns and anomalies over time. These dashboards help reviewers and security teams identify systemic trends, a particular agent consistently producing low-confidence outputs for a specific operation type, for example, that individual decision reviews miss.

Implementation steps

1. **Configure contextual grounding and automated reasoning:** Generate confidence scores for agent outputs through [Amazon Bedrock Guardrails](#) contextual grounding and automated reasoning, and include plain-language explanations alongside numeric scores in review notifications.

2. **Store historical decisions for similarity lookup:** Persist decisions and confidence scores in Amazon DynamoDB and implement a similarity query that surfaces past decisions for the same operation type alongside each new review request.
3. **Flag deviations from historical patterns:** When a confidence score deviates significantly from the historical average for that operation type, highlight it for the reviewer.
4. **Surface AgentCore Evaluations trends:** Integrate [Amazon Bedrock AgentCore Evaluations](#) quality scores into the review context so reviewers can see whether overall agent quality is stable or declining.
5. **Build dashboards for systemic trends:** Create Amazon Quick dashboards that visualize decision patterns, confidence score distributions, and anomaly trends over time for security team review.

Resources

Related best practices:

- [AGENTSEC04-BP01 Implement guardrails and alignment controls](#)
- [AGENTSEC07-BP01 Implement cognitive load management](#)
- [AGENTSEC07-BP04 Behavioral anomaly detection and agent containment](#)

Related documents:

- [Amazon Bedrock Guardrails documentation](#)
- [Amazon Bedrock AgentCore adds quality evaluations and policy controls](#)
- [Amazon Quick documentation](#)

Related services:

- [Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon DynamoDB](#)
- [Amazon Quick](#)
- [Amazon CloudWatch](#)

AGENTSEC07-BP03 Multiple reviewers for critical operations

A single reviewer is a single point of failure, both for honest errors and for social engineering. Independent, blind reviews for high-risk decisions are the defense-in-depth pattern, well known as the four-eyes principle, that keeps unilateral approval off the path.

Desired outcome:

- High-risk agent decisions receive independent review from multiple qualified reviewers, with blind review processes helping prevent anchoring bias.
- You resolve disagreements through escalation rather than defaulting to approval.
- You log all review decisions with reviewer identities and timestamps for audit purposes.

Common anti-patterns:

- Showing each reviewer the previous reviewer's decision before they submit their own, introducing anchoring bias that undermines independence.
- Defaulting to approval when reviewers disagree, letting a single approving reviewer effectively override a blocking reviewer.
- Assigning multiple reviews to reviewers from the same team or reporting chain, reducing the independence of the process.

Benefits of establishing this best practice:

- Multiple independent reviews provide defense-in-depth for human oversight, removing the single point of failure in the review process.
- Logged individual reviewer decisions, identities, and timestamps support compliance and enable investigation of approval anomalies.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Parallel execution orchestrates multi-reviewer workflows well. AWS Step Functions parallel execution branches send an independent review request to a different reviewer through Amazon SNS, and the workflow waits for all branches to complete before evaluating consensus. Blind

review comes from not including previous reviewer decisions in the notification content, so each reviewer evaluates the decision independently.

Consensus logic belongs in an AWS Lambda function that evaluates the collected decisions. Two-reviewer workflows require unanimous approval. Three or more reviewers use majority rules with escalation for split decisions, and escalation paths route disagreements to a senior reviewer with full visibility into individual decisions and their rationale.

Reviewer selection matters as much as the mechanism. Choose reviewers from different teams or organizational units to maximize independence. Reviewers who share a manager or work closely together tend to reach the same conclusion for social rather than analytical reasons. AWS IAM Identity Center manages reviewer identities so assignments are tracked and auditable.

Audit records live in Amazon S3 with reviewer identity, timestamp, decision (approve or reject), and optional rationale. Tag records with the associated agent operation ID to enable correlation with agent execution logs during investigations.

Implementation steps

1. **Orchestrate blind parallel reviews:** Design multi-reviewer workflows in AWS Step Functions with parallel branches, one per reviewer, that send independent blind review requests through Amazon SNS.
2. **Implement consensus and escalation:** Evaluate collected decisions in an AWS Lambda function, unanimous for two-reviewer flows, majority rules with escalation for three or more.
3. **Route split decisions to senior reviewers:** Configure escalation paths that give senior reviewers visibility into the individual decisions and rationale.
4. **Select reviewers from different teams:** Use AWS IAM Identity Center to manage reviewer identities and track assignments, and draw reviewers from different organizational units.
5. **Persist decisions to S3:** Store all review decisions in Amazon S3 with reviewer identity, timestamp, and decision rationale, tagging records with the agent operation ID for correlation with execution logs.

Resources

Related best practices:

- [AGENTSEC04-BP02 Human-in-the-loop for critical decisions](#)
- [AGENTSEC07-BP01 Implement cognitive load management](#)

- [AGENTSEC07-BP04 Behavioral anomaly detection and agent containment](#)

Related documents:

- [AWS Step Functions parallel states](#)
- [Human approval in Step Functions](#)
- [Implement human-in-the-loop confirmation with Amazon Bedrock Agents](#)

Related services:

- [AWS Step Functions](#)
- [Amazon SNS](#)
- [Amazon S3](#)
- [AWS Lambda](#)
- [AWS IAM Identity Center](#)

AGENTSEC07-BP04 Behavioral anomaly detection and agent containment

Detection without containment leaves issues identified but running. Containment without detection relies on manual observation. Per-agent baselines paired with automated credential revocation and forensic capture stop affected agents within minutes while preserving what investigators need.

Desired outcome:

- You establish behavioral baselines per agent and trigger real-time alerts when deviations cross defined thresholds.
- You automatically isolate agents exhibiting anomalous behavior within minutes of detection through credential revocation and circuit breaker activation.
- You capture forensic state before isolation, and manual override capabilities allow incident responders to quarantine or restore agents when human judgment is required.

Common anti-patterns:

- Monitoring only infrastructure metrics without agent-specific behavioral indicators, missing signals such as API call patterns, decision frequencies, and data access volumes.
- Deploying anomaly detection without establishing behavioral baselines first, producing excessive false positives or missed detections.
- Relying on manual quarantine processes that require human intervention, letting an affected agent continue operating for hours while waiting for human response.
- Implementing quarantine by stopping the agent process without revoking credentials, so the agent can be restarted with the same identity and permissions.
- Not preserving agent state and logs before quarantine, losing forensic evidence from the agent's memory, active sessions, and pending operations.

Benefits of establishing this best practice:

- Automated credential revocation and circuit breaker activation isolate affected agents within minutes of detection.
- Forensic preservation through state capture before isolation provides evidence for investigation without relying on the agent's own logs.
- Circuit breakers route dependent workflows to safe fallback paths rather than allowing cascading failures.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Behavioral signals differ by agent type. A RAG agent and a coding agent have very different normal patterns, so generic thresholds produce either false alarms or missed detections. Start with the general categories (API call rate, data access volume, decision frequency, error rate, resource consumption) but pick the specific measurements that actually make sense for each agent. A customer-support agent might track outbound email volume and cross-customer retrievals, a coding agent might track commit rates and commands executed, and a data-analysis agent might track query volume and cross-table joins. Amazon CloudWatch anomaly detection on the selected metrics establishes dynamic baselines that adapt to normal variation patterns, reducing false positives compared to static thresholds, and alarms fire when metrics deviate beyond the anomaly detection band.

[Amazon Bedrock AgentCore Evaluations](#) adds a detection layer at the behavior-quality level. Built-in evaluators (correctness, tool-selection accuracy, helpfulness, safety) are a starting point, but custom evaluators capture the quality dimensions that matter for your agent, whether outputs conform to your organization's policy, whether the agent is using the expected tools for its domain, and whether it accomplishes the task it was assigned. A sudden drop in evaluation scores serves as an early-warning signal that behavior is drifting before infrastructure-level anomaly alarms fire. Amazon CloudWatch alarms on evaluation scores alongside behavioral metrics give you layered detection.

Two AWS security services add complementary signals you don't need to instrument yourself. Amazon GuardDuty analyzes AWS CloudTrail, VPC flow logs, and DNS logs to detect anomalous API call patterns for IAM roles (unexpected regions, unusual service combinations, known-malicious IPs), which catches agent behavior CloudWatch metrics would miss unless you explicitly measured it. Amazon Macie inspects Amazon S3 objects and access patterns for sensitive-data exposure (agent-accessed buckets containing unusual volumes of PII or credentials), which is orthogonal to API-level metrics. AWS Security Hub CSPM centralizes CloudWatch anomaly alarms, GuardDuty, and Macie findings so one source's anomaly can be correlated with the others during investigation rather than treated in isolation.

When anomaly detection triggers above a defined severity threshold, Amazon EventBridge rules invoke either an AWS Lambda function or an AWS Systems Manager Automation document. Lambda fits containment logic with custom code paths, external API calls, or conditional branching that benefits from full programming flexibility. SSM Automation fits when the containment sequence is a series of well-defined steps (native step definitions, parameters, and rollback without code) and you want the same runbook pattern for automatic and manual containment. Either way, the sequence runs in this order: capture a forensic snapshot of the agent's current memory, active sessions, and pending operations to Amazon S3, then revoke the agent's credentials by attaching a deny-all policy to its IAM role (preserving the role for forensic analysis), then broadcast a quarantine event through Amazon EventBridge to notify dependent workflows to activate their circuit breaker logic.

Circuit breakers in AWS Step Functions workflows that depend on quarantinable agents handle the downstream impact. Catch states detect agent unavailability and route workflow execution to a safe fallback path rather than failing with an unhandled error. A manual override interface through AWS Systems Manager Automation runbooks lets incident responders quarantine or restore agents through a controlled, auditable process, and multi-person authorization for restoration helps prevent premature re-activation.

Implementation steps

1. **Choose agent-specific metrics and baseline them:** Pick meaningful metrics for each agent from the general categories, configure Amazon CloudWatch anomaly detection on them, and establish baselines during normal operation.
2. **Add evaluation-based early warning:** Deploy [Amazon Bedrock AgentCore Evaluations](#) with built-in and custom evaluators, and configure Amazon CloudWatch alarms on evaluation scores.
3. **Centralize security findings:** Enable Amazon GuardDuty and Amazon Macie for all agent accounts and centralize findings in AWS Security Hub CSPM.
4. **Automate containment on threshold exceedance:** Implement Amazon EventBridge rules that invoke AWS Lambda or AWS Systems Manager Automation when anomaly severity exceeds thresholds, and sequence forensic capture, credential revocation, and quarantine event broadcast.
5. **Wire circuit breakers into dependent workflows:** Configure catch states in AWS Step Functions workflows that depend on quarantinable agents, routing to safe fallback paths on agent unavailability.
6. **Provide a manual runbook with multi-person auth:** Create AWS Systems Manager Automation runbooks for manual quarantine and restoration with multi-person authorization required for restoration.
7. **Test quarterly:** Run containment procedure tests every quarter to validate isolation, circuit breakers, and forensic capture.

Resources

Related best practices:

- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)
- [AGENTSEC06-BP02 Implement workflow orchestration security controls](#)
- [AGENTSEC06-BP04 Monitor and detect coordination anomalies](#)
- [AGENTREL02-BP03 Implement behavioral anomaly detection and monitoring](#)
- [AGENTREL07-BP02 Enable automatic recovery from agent execution failures](#)

Related documents:

- [Amazon CloudWatch anomaly detection](#)

- [Amazon Bedrock AgentCore adds quality evaluations and policy controls](#)
- [Amazon GuardDuty documentation](#)
- [Amazon Macie documentation](#)
- [AWS Step Functions error handling](#)

Related services:

- [Amazon CloudWatch](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon GuardDuty](#)
- [Amazon Macie](#)
- [AWS Security Hub CSPM](#)
- [Amazon EventBridge](#)
- [AWS Lambda](#)
- [AWS Step Functions](#)
- [AWS Systems Manager](#)
- [AWS Identity and Access Management](#)

AGENTSEC07-BP05 Regular security assessments and red teaming

Untested security controls degrade quietly as techniques evolve and configurations drift. A combination of continuous automated scanning and periodic human-led red team exercises validates that guardrails, detection rules, and response procedures still work against current attacks.

Desired outcome:

- You run automated security scanning continuously against agent deployments (on each deploy and on schedule) and conduct human-led red team exercises on a regular cadence with scenarios targeting agent manipulation, including prompt injection, goal hijacking, and tool misuse.
- You document findings, track them to remediation, and use them to update security controls and detection rules.

Common anti-patterns:

- Conducting generic application security assessments without agent-specific scenarios, missing prompt injection, memory poisoning, and multi-agent coordination issues that traditional testing doesn't cover.
- Performing red team exercises only at initial deployment without scheduling regular assessments, missing techniques that emerge as the threat environment evolves.
- Not tracking findings to remediation, letting identified issues persist so the assessment produces work but no posture improvement.
- Conducting red team and blue team activities in isolation without purple team collaboration, limiting the knowledge transfer that improves detection and response.

Benefits of establishing this best practice:

- Realistic testing confirms guardrails, detection rules, and response procedures work against current techniques.
- Assessment findings drive updates to guardrail configurations, permission boundaries, and detection rules in a continuous feedback loop.
- Purple team activities transfer knowledge from red to blue teams, improving the organization's ability to detect and respond to agent-specific issues.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The assessment cadence needs to match the risk level of each agent deployment. Automated scanning runs continuously, and manual red team exercises run on a schedule. For automated agentic testing, [AWS Security Agent](#) provides on-demand penetration testing that executes attack chains adapted to the target application, covering prompt injection, jailbreaking, goal hijacking, and related patterns (see AGENTSEC09-BP02 for integrating it into broader penetration testing workflows). Supplement automated testing with manual exercises that explore novel scenarios specific to your agent architecture.

Red team scenarios need structure. The [OWASP Top 10 for Agentic Applications \(2026\)](#) covers agent manipulation risks specifically, prompt injection, tool misuse, identity and privilege abuse, and agent behavior hijacking, and the [OWASP Top 10 for LLM Applications](#) covers model-level risks that still apply. Build a scenario library that covers multi-agent coordination issues, memory

poisoning, tool misuse, and human-in-the-loop bypass techniques, and document each scenario with description, expected detection mechanism, and success criteria.

[Amazon Bedrock AgentCore Evaluations](#) supports the assessment process by providing a continuous quality baseline. Running evaluations before and after red team exercises measures whether the exercise exposed quality degradation that the existing evaluators did not catch, and the results refine custom evaluator prompts and scoring thresholds.

Durable, versioned storage keeps the historical record intact. Store scenarios, execution results, and remediation tracking in Amazon S3 with versioning enabled, or in a dedicated test management system that maintains change history. Map red team findings to your compliance control framework so assessment results produce audit evidence consistent with your regulatory requirements.

Purple team activities close the loop. Bringing red team and blue team together to review scenarios and detection responses updates Amazon CloudWatch alarms, Guardrails configurations, and incident response runbooks based on observed patterns. Tracking improvements in detection time and response effectiveness across cycles demonstrates the program's value.

Implementation steps

1. **Establish an assessment schedule:** Set a cadence appropriate for each agent deployment's risk level.
2. **Build a scenario library:** Develop red team scenarios based on the OWASP Top 10 for Agentic Applications (primary) and the OWASP Top 10 for LLM Applications (supplementary), covering prompt injection, memory poisoning, tool misuse, and HITL bypass.
3. **Integrate automated agentic testing:** Deploy agentic AI red teaming tools and include them in the assessment workflow for automated coverage of common patterns.
4. **Measure quality impact before and after:** Run [Amazon Bedrock AgentCore Evaluations](#) before and after red team exercises to measure quality impact and refine evaluator configurations.
5. **Persist and map findings:** Store scenarios, results, and remediation tracking in durable, versioned storage (Amazon S3 with versioning enabled), and map findings to your compliance control framework for audit evidence.
6. **Run purple team sessions:** Update detection rules, guardrail configurations, and incident response runbooks based on each assessment cycle's findings.

Resources

Related best practices:

- [AGENTSEC04-BP01 Implement guardrails and alignment controls](#)
- [AGENTSEC07-BP04 Behavioral anomaly detection and agent containment](#)
- [AGENTSEC08-BP01 Multi-layer input validation and prompt injection defense](#)

Related documents:

- [OWASP Top 10 for Agentic Applications \(2026\)](#)
- [OWASP Top 10 for Large Language Model Applications](#)
- [AWS Security Agent](#)
- [Responsible AI in action: How Data Reply red teaming supports generative AI safety on AWS](#)
- [Protect DeepSeek model deployments with Protect AI and Amazon Bedrock](#)
- [Amazon Bedrock AgentCore adds quality evaluations and policy controls](#)

Related services:

- [Amazon S3](#)
- [Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Secure agent inputs and outputs

Agents process inputs from users, tools, and other agents, and they generate outputs for downstream systems, memory stores, and end users. Without proper validation, adversarial inputs such as prompt injection can influence agent behavior, and agent outputs can inadvertently disclose sensitive information or generate harmful content. Multi-layer validation at both the input and output boundaries helps verify that agents operate safely within their intended scope.

AGENTSEC08: How do you validate and secure agent inputs and outputs?

Capability intent

- Every agent input surface (direct user messages, tool outputs, inter-agent messages, retrieved external content, and memory reads) has validation appropriate to its risk profile, with no input reaching the agent's reasoning process unchecked.
- Indirect prompt injection through retrieved documents, web pages, and API responses is treated as a first-class risk and filtered with the same rigor applied to direct user inputs.
- Agent outputs are inspected for personally identifiable information (PII), credentials, and other sensitive data on every outbound path (user responses, inter-agent messages, memory writes, and audit logs), with consistent masking or blocking applied at each boundary.
- Validation strictness is calibrated to the likelihood and impact of each risk scenario rather than applied uniformly, and telemetry on confidence scores, blocked content, and false-positive rates drives continual tuning.
- Filtering and guardrail decisions are logged with enough context to support compliance auditing, to detect systemic data leakage patterns, and to feed periodic re-optimization of denied topics, sample prompts, and filter thresholds.

Maturity levels

These levels summarize what each stage of maturity looks like for secure agent inputs and outputs as a whole.

Level	Name	What it looks like
1	Initial	Input validation, if present, runs only on direct user messages. Retrieved external content, tool outputs, inter-agent messages, and memory reads are treated as trusted. Sensitive information in agent outputs is controlled, if at all, by asking the model to self-censor. Prompt injection and data leakage incidents are found by users or downstrea

Level	Name	What it looks like
		m systems rather than by controls.
2	Emerging	Amazon Bedrock Guardrails is deployed with prompt attack detection and basic sensitive information filters covering common PII categories, primarily on user inputs and user-facing responses. Denied topics are defined but not tuned against representative data, and assessment modes use default settings. Inter-agent messages, memory writes, and tool outputs are covered inconsistently.

Level	Name	What it looks like
3	Defined	Every agent has its input surfaces mapped and a validation control assigned to each, applied through the ApplyGuardrail API with prompt attack detection in block mode. Output filtering runs on every outbound path, including inter-agent messages, writes to Amazon Bedrock AgentCore Memory , and audit logs, with masking or blocking chosen per data classification. Guardrail telemetry captures per-filter confidence scores and blocked content in Amazon CloudWatch for review.

Level	Name	What it looks like
4	Proactive	Guardrails configuration is tuned periodically using the Amazon Bedrock Guardrails Optimizer against annotated production samples. Denied topic definitions use the full character budget with representative sample prompts, and assessment modes are selected per filter category based on likelihood and impact. Amazon Comprehend custom entity recognizers extend coverage to organization-specific data types, and filtering decisions feed alarms that detect systemic increases in sensitive-data generation or blocked-content volume. Compliance auditing is in place: guardrail and filtering decisions are written to tamper-evident storage with the input, output, intervention type, filter scores, and the originating agent and user identity, retention is aligned to the applicable regulatory framework, and standard audit queries support reporting against frameworks such as the EU AI Act, SOC 2, and HIPAA.

Level	Name	What it looks like
5	Optimized	Input validation and output filtering are self-tuning: optimization runs on a schedule against fresh production samples, false-positive and false-negative rates are primary metrics, and validation telemetry is correlated with runtime threat detection and penetration-test findings to identify active exploitation. Content retrieval is restricted to approved sources with runtime attestation, and evolving attack patterns flow directly into updated guardrails, denied topics, and filter thresholds.

Common issues to watch for

- Input and output boundaries receive uneven defense, with rigorous checks on user inputs but little coverage for tool outputs, retrieved external content, inter-agent messages, and memory reads, leaving the most commonly targeted vectors unprotected.
- Guardrails are treated as a one-time configuration rather than a tuning loop, so denied topic definitions, filter thresholds, and assessment modes decay as attack patterns and data classifications change.
- Guardrail and filtering decisions are not captured with enough context to tell when validation is degrading or which categories need retuning, so quality issues stay invisible until an incident surfaces them.
- Sensitive information controls lean on the model to self-censor rather than on enforced output filtering, so PII and credentials pulled in from tool outputs or retrieved documents flow through whenever the model repeats them.

- Filter rules are either overly broad, which erodes user trust and pushes teams to weaken or disable them, or overly narrow, which leaves gaps. Neither side has a feedback loop against annotated outcomes to calibrate strictness by risk scenario.

Best practices

- [AGENTSEC08-BP01 Multi-layer input validation and prompt injection defense](#)
- [AGENTSEC08-BP02 Output filtering for sensitive information](#)

AGENTSEC08-BP01 Multi-layer input validation and prompt injection defense

Agents take input from many surfaces and only one needs to be unvalidated for adversarial content to reach the agent's reasoning process. A layered validation architecture covers every surface, and in particular catches the indirect prompt injection embedded in retrieved external content.

Desired outcome:

- Every input surface has a validation layer appropriate to its risk profile, and no input reaches the agent's reasoning process without passing through at least one validation control.
- You specifically address indirect prompt injection through retrieved external content, which is the surface most commonly missed when validation is applied only to direct user inputs.

Common anti-patterns:

- Applying input validation only to direct user inputs while skipping validation for data retrieved from external sources, letting embedded instructions in web pages, documents, and API responses bypass user-facing validation.
- Validating at one input surface but not others (for example, validating user inputs with Guardrails but not validating tool outputs before they enter the agent's context), creating gaps that can be targeted.
- Defining denied topics with vague or overly broad descriptions that generate false positives on legitimate content, eroding trust and prompting teams to weaken or disable guardrails entirely.

Benefits of establishing this best practice:

- Defense-in-depth architecture where each input surface has validation appropriate to its risk profile helps cover every surface.
- Validation of external content before it enters the agent's context closes the most commonly missed gap: indirect prompt injection.
- Confidence-based assessment modes let organizations tune validation strictness per filter category based on the likelihood and impact of each risk scenario.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Input surfaces to an agent are not one thing. Direct user messages, tool outputs, inter-agent messages, retrieved external content (web pages, documents, API responses), and memory reads are all paths by which data reaches the agent's context, and each needs a validation control. Surface-specific guidance lives in AGENTSEC01-BP02 (memory inputs), AGENTSEC02-BP02 (tool parameters), and AGENTSEC04-BP01 (goal alignment guardrails). This best practice is the architectural framing and focuses on the cross-cutting concern those others don't cover: indirect prompt injection through external content retrieval.

External content retrieval is the most commonly missed surface. When an agent uses RAG, web browsing, or API calls to gather information during task execution, the retrieved content becomes part of the agent's context, and adversarial instructions embedded in that content (indirect prompt injection) influence the agent's behavior as effectively as a direct user injection. Apply [Amazon Bedrock Guardrails](#) with prompt attack detection to all retrieved content before it enters the agent's context, and implement content source validation that restricts the agent to retrieving content from approved domains or data sources where feasible.

Guardrails provides a unified validation mechanism that can be applied across multiple input surfaces through the ApplyGuardrail API. Configure a guardrail with prompt attack detection, denied topics, and word filters once, and apply it at each input boundary. That gives you consistent policy enforcement across surfaces with surface-specific tuning through guardrail versioning.

Two assessment modes matter: block mode returns a binary allow or deny decision, and detect mode returns confidence scores for each filter category without blocking the request. Use block mode for prompt attack detection, where even low-confidence matches warrant intervention given the severity of potential impact. For content safety filters on internal or lower-risk applications, detect mode lets the application make risk-proportionate decisions based on the confidence scores

returned. Score each risk scenario by likelihood and impact to determine appropriate confidence thresholds per filter category rather than applying uniform thresholds.

Denied topics use probabilistic, LLM-based evaluation to determine whether content matches a topic definition, and definition quality drives accuracy. Use the full 1,000-character limit for each denied topic definition with specific and unambiguous descriptions, and populate all five sample prompt fields (up to 200 characters each) with representative examples that illustrate the boundary between restricted and permitted content. Vague or broad definitions inflate false positive rates, which erodes user trust and pressures teams to weaken or disable guardrails.

When using the ApplyGuardrail API directly (rather than through the Converse API or Amazon Bedrock Agents), guardrail assessment results are not automatically published to Amazon CloudWatch. You are responsible for the telemetry pipeline that captures assessment outcomes, confidence scores, and blocked content. Set the outputScope parameter to full on ApplyGuardrail API calls to receive complete assessment data including per-filter confidence scores, which are essential for adjusting thresholds and feeding the Guardrails Optimizer. Log both the request content and the assessment response for blocked items, this data is required for ongoing configuration refinement and false-positive analysis.

The Amazon Bedrock Guardrails Optimizer is a reference implementation on AWS Samples that automates guardrail configuration refinement. It uses a Strands Agent to iteratively adjust denied topic definitions, sample prompts, and filter thresholds based on annotated test data. As opposed to model fine-tuning, this is policy configuration optimization. The agent analyzes failed test cases, rewrites the guardrail configuration, re-evaluates against the test dataset, and repeats until target accuracy is reached. Prepare a representative dataset annotated with expected outcomes (allow or deny for each filter category), run the Optimizer during initial guardrail setup, and schedule periodic re-runs (monthly or quarterly) using samples from production traffic to adapt to evolving content patterns and reduce false positive rates over time.

Implementation steps

1. **Map input surfaces and assign controls:** Identify all input surfaces for each agent and the validation control covering each surface, flagging any that are currently unvalidated.
2. **Validate retrieved external content:** Configure [Amazon Bedrock Guardrails](#) with prompt attack detection and apply it to external content (RAG results, web content, API responses) before it enters the agent's context.

3. **Pick assessment mode by risk:** Use block mode for prompt attack detection filters and consider detect mode for content safety filters on lower-risk applications, implementing application-level logic to make risk-proportionate decisions based on returned confidence scores.
4. **Write precise denied topics:** Use the full 1,000-character limit for each denied topic definition and populate all five sample prompt fields with representative examples that illustrate the boundary between restricted and permitted content.
5. **Capture ApplyGuardrail telemetry:** Set outputScope to full on all ApplyGuardrail API calls and implement a telemetry pipeline to capture assessment outcomes, confidence scores, and blocked content in Amazon CloudWatch.
6. **Run the Guardrails Optimizer:** Run the Amazon Bedrock Guardrails Optimizer with an annotated test dataset during initial setup, then schedule periodic re-optimization (monthly or quarterly) using samples from production traffic.
7. **Restrict content sources where feasible:** Implement content source validation that restricts agents to retrieving content from approved domains or data sources.
8. **Verify surface-specific controls exist:** Confirm that the controls described in AGENTSEC01-BP02 (memory inputs), AGENTSEC02-BP02 (tool parameters), and AGENTSEC04-BP01 (goal alignment guardrails) are implemented for each applicable agent.
9. **Log and review blocked inputs:** Log all blocked inputs across surfaces and review patterns periodically to identify new techniques and surfaces that may need additional coverage.

Resources

Related best practices:

- [AGENTSEC01-BP02 Validate and sanitize memory inputs](#)
- [AGENTSEC02-BP02 Validate tool inputs and outputs](#)
- [AGENTSEC04-BP01 Implement guardrails and alignment controls](#)
- [AGENTSEC08-BP02 Output filtering for sensitive information](#)

Related documents:

- [Amazon Bedrock Guardrails prompt attack detection](#)
- [Amazon Bedrock Guardrails denied topics](#)
- [Amazon Bedrock Guardrails ApplyGuardrail API reference](#)

- [Build responsible AI applications with Amazon Bedrock Guardrails](#)
- [Best practices with Amazon Bedrock Guardrails filter configuration](#)
- [Amazon Bedrock AgentCore Memory best practices](#)

Related examples:

- [Amazon Bedrock Guardrails Evaluation and Optimization Framework](#)

Related services:

- [Amazon Bedrock Guardrails](#)

AGENTSEC08-BP02 Output filtering for sensitive information

Filtering user-facing responses leaves internal data paths (inter-agent messages, memory writes, audit logs) as open channels for PII and credentials to escape. Scanning every output path with a data classification policy keeps sensitive content inside the agent's authorized handling scope.

Desired outcome:

- You scan agent outputs for PII, credentials, and other sensitive data before returning them to users or downstream systems, with content masked or blocked based on data classification policies.
- Agent outputs containing credentials, private keys, or regulated PII are blocked or masked before they reach end users or external systems.
- You log output filtering decisions for compliance auditing.

Common anti-patterns:

- Relying on the model to self-censor sensitive information, when models do generate outputs containing PII or credentials (especially when that information is in the agent's context from tool outputs or retrieved documents).
- Applying output filtering only to user-facing responses while skipping filtering for outputs passed to other agents or stored in memory, creating data-leakage paths through internal communications.

- Using overly broad masking rules that mask legitimate content alongside sensitive data, degrading output quality to the point users work around the filter.

Benefits of establishing this best practice:

- Data classification enforcement at the agent boundary helps keep sensitive information within the agent's authorized data-handling scope regardless of what the model generates.
- Logged filtering decisions support compliance with data protection regulations.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Every output path is a potential exfiltration path. User-facing responses are the obvious one, but an agent also writes to memory, passes content to other agents, and emits logs. Filtering only the user-facing path leaves the others as open channels for sensitive data, and adversarial inputs designed to exfiltrate data don't consistently target the user-facing channel. The architectural requirement is that every output surface passes through the same filter, not just the ones users see.

[Amazon Bedrock Guardrails](#) sensitive information filters cover the data types relevant to most use cases:

- PII categories (names, addresses, phone numbers, email addresses, SSNs, and financial account numbers)
- Credentials (API keys, passwords, and private keys)
- Custom entity types specific to your organization

Configure the filter action as MASK for most sensitive data types to preserve output utility while protecting sensitive content, and BLOCK for the most sensitive categories such as credentials and private keys where masking isn't enough.

Apply the filter as middleware in the agent output pipeline, so every output destination, user-facing responses, inter-agent messages, [Amazon Bedrock AgentCore Memory](#) writes through the `create_event` API, and audit logs, flows through it. AgentCore Memory's built-in long-term memory strategies already filter PII from extracted long-term records by default, but short-term memory (raw events) retains original content, so compliance requirements that prohibit PII in any stored

form require applying Guardrails sensitive information filters before writing events to short-term memory as well.

For organization-specific sensitive data types not covered by built-in categories, Amazon Comprehend custom entity recognizers extend coverage. Train recognizers on examples of your sensitive data types and integrate them into the output filtering pipeline through AWS Lambda functions that call the Amazon Comprehend API before returning outputs.

Logging every filtering decision (the type of sensitive data detected, the action taken (mask or block), the output destination) produces the data loss prevention report and catches patterns where the agent is systematically generating outputs containing sensitive data (a signal of prompt injection or data exfiltration). Amazon CloudWatch alarms on elevated detection rates turn that signal into an active alert.

Implementation steps

1. **Configure sensitive information filters:** Set up [Amazon Bedrock Guardrails](#) with filters for all relevant PII and credential categories, choosing MASK or BLOCK per category based on data classification policy.
2. **Filter every output path:** Apply output filtering as a middleware layer for user-facing responses, inter-agent messages, memory writes, and audit logs.
3. **Filter before short-term memory writes when required:** For compliance requirements that prohibit PII in any stored form, apply Guardrails filtering before writing events to [Amazon Bedrock AgentCore Memory](#) short-term storage.
4. **Extend with Amazon Comprehend custom entities:** Train Amazon Comprehend custom entity recognizers for organization-specific sensitive data types and integrate them into the filtering pipeline.
5. **Log and alarm on detections:** Log all output filtering decisions with data type, action, and destination metadata, and configure Amazon CloudWatch alarms for elevated sensitive data detection rates.
6. **Review configurations periodically:** Adjust output filtering to match evolving data classification policies and new regulated data types.

Resources

Related best practices:

- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)
- [AGENTSEC08-BP01 Multi-layer input validation and prompt injection defense](#)

Related documents:

- [Amazon Bedrock Guardrails sensitive information filters](#)
- [Amazon Bedrock AgentCore Memory best practices](#)
- [Amazon Comprehend documentation](#)

Related services:

- [Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon Comprehend](#)
- [Amazon CloudWatch](#)
- [AWS Lambda](#)

Agent vulnerability scanning and penetration testing

Agentic AI systems introduce new attack surfaces that traditional application security testing doesn't cover. Agents reason about inputs, chain multi-step actions, interact with tools and APIs, and operate with varying degrees of autonomy. These behaviors create complex vulnerability paths that rule-based scanners and periodic manual assessments can't address. Automated vulnerability scanning and penetration testing tailored to agentic workloads help you identify exploitable weaknesses across the full development lifecycle, from design documents and code through to running applications, and validate that security controls hold up under realistic attack scenarios.

AGENTSEC09: How do you perform vulnerability scanning and penetration testing for agentic AI systems?

Capability intent

- Security testing covers the full agentic AI development lifecycle, from design documents through pull requests and into running applications, so vulnerabilities are caught at the point where remediation is cheapest.
- Vulnerability scanning and penetration testing reason about agent behavior, including tool invocations, multi-agent delegation, memory handling, and prompt injection chains, rather than relying only on signatures for known web application flaws.
- Findings are validated through actual exploitation, ranked by real-world exploitability, and paired with reproducible attack paths and ready-to-implement fixes so that development teams can remediate without waiting for specialist intervention.
- Security assessments run in dedicated testing environments with scoped credentials, logged activity, and impact-containment controls, so realistic attack simulation doesn't put production agents, memory, or downstream systems at risk.
- Runtime threat detection, vulnerability scanning, and penetration test findings are correlated in a unified view, with automated remediation workflows that tighten mean time to detection and mean time to remediation.

Maturity levels

These levels summarize what each stage of maturity looks like for agent vulnerability scanning and penetration testing as a whole.

Level	Name	What it looks like
1	Initial	Security scanning is limited to rule-based static analysis against known vulnerability signatures, run late in the development lifecycle or only before release. Penetration testing, if it happens, targets generic web application flaws and doesn't exercise agent-specific attack surfaces. Testing runs against shared

Level	Name	What it looks like
		or production-like environments with long-lived credentials, and findings are delivered without reproducible attack paths or fix guidance.
2	Emerging	Design documents and pull requests receive security review, often manually, and a baseline scanner runs in the build pipeline. Penetration testing uses a documented scenario library but runs on a periodic cadence that lags agent capability changes. Test environments are separated from production for high-risk assessments, and test credentials are stored in AWS Secrets Manager . Findings are tracked to remediation but the handoff between security and development teams is still manual.

Level	Name	What it looks like
3	Defined	AI-powered vulnerability scanning runs across design, development, and deployment phases, using tools such as AWS Security Agent to evaluate architecture documents, pull requests, and running applications against organization-wide security requirements. On-demand penetration testing exercises agent orchestration endpoints, tool invocation paths, and multi-agent communication channels, with findings carrying CVSS scores and reproducible attack paths. Test environments replicate production behavior with isolated memory stores and scoped credentials, and findings from Amazon GuardDuty , Amazon Macie , and Amazon Inspector are aggregated in AWS Security Hub CSPM .

Level	Name	What it looks like
4	Proactive	Security validation is integrated into CI/CD and runs whenever agent system prompts, tool registrations, permission scopes, orchestration patterns, or code are modified. AWS Security Agent delivers code fix suggestions alongside validated findings, and regression testing confirms that fixes are effective. Amazon GuardDuty Extended Threat Detection correlates multi-step attack sequences across API activity, network behavior, and data access. Automated remediation workflows capture forensic state, apply containment, and route fixes through a tracked pipeline. Amazon CloudWatch composite alarms combine Amazon Bedrock AgentCore evaluation score drift with runtime threat findings for high-confidence detection.

Level	Name	What it looks like
5	Optimized	Vulnerability scanning, penetration testing, runtime threat detection, and agent evaluations operate as a single feedback loop that continuously refines detection logic, remediation automation, and scenario coverage. Penetration testing runs on-demand and feeds measurable improvement in mean time to detection and mean time to remediation. Test environments, scoped credentials, and impact-containment controls are validated through controlled failure injection, and purple team outcomes flow back into scanner heuristics and detection rules.

Common issues to watch for

- Security testing concentrates on deployment-time scans of a production build, skipping design-phase review and pull-request analysis, so architectural flaws and agent-behavior issues are found late when remediation is most expensive.
- Scanners and pentests are adapted from traditional web applications and don't exercise agent-specific surfaces such as tool parameter manipulation, memory poisoning, multi-agent trust boundaries, and prompt injection chains, leaving the most distinctive agentic risks untested.
- Test activities run against production agents, shared memory stores, or downstream systems because no scoped testing environment exists, and realistic attack simulation is avoided to protect live data rather than because controls are known to work.

- Vulnerability findings reach development teams without validated exploit paths, severity context, or suggested fixes, so remediation is slow, inconsistent, and often addresses symptoms rather than the underlying issue.
- Pre-deployment scans, penetration test findings, and runtime threat signals live in separate tools with no correlation, so known vulnerabilities that are being actively exploited are not prioritized ahead of theoretical ones.

Best practices

- [AGENTSEC09-BP01 Integrate AI-powered vulnerability scanning across the development lifecycle](#)
- [AGENTSEC09-BP02 Conduct context-aware penetration testing with multi-agent attack simulation](#)
- [AGENTSEC09-BP03 Implement continuous security validation with automated remediation](#)
- [AGENTSEC09-BP04 Establish scoped and controlled testing environments for agent security assessments](#)
- [AGENTSEC09-BP05 Implement runtime threat detection, security event correlation, and automated remediation for agents](#)

AGENTSEC09-BP01 Integrate AI-powered vulnerability scanning across the development lifecycle

Pattern-matching scanners find common bugs but miss context-dependent flaws in agent orchestration, tool interactions, and authorization chains. AI-powered scanning that reasons about code and design documents the way a human security researcher does catches these issues at the phase when remediation is cheapest.

Desired outcome:

- Vulnerability scanning is embedded at every phase of the agentic AI development lifecycle, covering design documents, pull requests, and deployed applications.
- You review design documents for security risks before code is written, analyze pull requests for common and agent-specific vulnerabilities during development, and continually scan deployed applications for emerging threats.
- Findings carry severity ratings, confidence scores, and practical remediation guidance so teams can prioritize and fix issues efficiently.

Common anti-patterns:

- Relying on rule-based static analysis that matches known vulnerability patterns, missing context-dependent issues in agent orchestration logic, insecure tool parameter handling, or broken access control in multi-agent delegation chains.
- Performing security scanning only at deployment time rather than across design and development phases, letting vulnerabilities accumulate and making late-discovery remediation expensive.
- Treating AI-generated code the same as human-written code for security review, ignoring the distinct vulnerability patterns AI coding assistants introduce (hallucinated API calls, insecure default configurations, and outdated library usage).

Benefits of establishing this best practice:

- Design-phase security reviews identify architectural risks before code is written, reducing remediation cost and development delays.
- AI-powered scanning that reasons about application context and agent behavior catches complex vulnerabilities that pattern-matching tools miss.
- Automated scanning integrated into CI/CD pipelines scales security expertise across development teams without creating bottlenecks.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Pattern matching against known signatures isn't enough for agentic systems. A SQL injection signature doesn't catch a broken multi-agent authorization chain, and a missing-input-validation rule doesn't catch a tool parameter manipulated through a prompt injection. The scanning that reaches those issues has to reason about how components interact, trace data flows through the orchestration layer, and understand what the agent was actually intended to do. That is the capability AI-powered scanning adds, and why it is effective on agent-specific flaws.

Deploy scanning across the full lifecycle. At the design phase, use tools that analyze architecture documents, product specifications, and technical designs for security risks before code is written. During development, integrate scanning into code review workflows to analyze pull requests for both common vulnerabilities (SQL injection, missing input validation) and agent-specific issues

(insecure tool invocations, insufficient permission scoping). At deployment, run on-demand scans against running applications to validate that security controls hold under realistic conditions.

[AWS Security Agent](#) provides this lifecycle coverage as a single capability. Security teams define organizational security requirements once in the AWS Management Console (approved authorization libraries, logging standards, data access policies), and AWS Security Agent enforces them throughout development, evaluating design documents and code against the standards and providing specific guidance when it detects violations. For code security reviews, configure AWS Security Agent to monitor repositories and analyze pull requests so evaluation scales across codebases while keeping oversight on critical issues.

AI-generated code needs extra scrutiny. AI coding assistants introduce vulnerability patterns that differ from typical human-written code (hallucinated API calls, insecure default configurations, outdated dependency usage), and scanning tools need to flag these explicitly. Tools like Claude Code Security use multi-stage verification where findings are re-examined to prove or disprove results and filter out false positives before they reach analysts, which reduces noise and lets teams focus on validated issues.

Implementation steps

1. **Codify security requirements centrally:** Define organizational security requirements (approved libraries, logging standards, data access policies) and configure them in [AWS Security Agent](#) for automated enforcement across development teams.
2. **Run design-phase reviews:** Configure AWS Security Agent to analyze architecture documents and technical specifications before development begins.
3. **Enable PR-level code review:** Connect AWS Security Agent to your code repositories to cover both human-written and AI-generated code on every pull request.
4. **Configure multi-stage verification:** Set up AI-powered scanning with multi-stage verification to reduce false positives and assign severity ratings to validated findings.
5. **Triage and track to resolution:** Route validated vulnerabilities to the appropriate team with remediation guidance, and track findings through to resolution.

Resources

Related best practices:

- [AGENTSEC07-BP05 Regular security assessments and red teaming](#)

- [AGENTSEC08-BP01 Multi-layer input validation and prompt injection defense](#)
- [AGENTSEC09-BP02 Conduct context-aware penetration testing with multi-agent attack simulation](#)

Related documents:

- [AWS Security Agent](#)
- [Security Considerations for AWS Security Agent and AI assisted penetration testing](#)
- [Claude Code Security, Making frontier cybersecurity capabilities available to defenders](#)
- [OWASP Top 10 for LLM Applications](#)

Related services:

- [Amazon Bedrock Guardrails](#)
- [Amazon CodeGuru Security](#)

AGENTSEC09-BP02 Conduct context-aware penetration testing with multi-agent attack simulation

Generic scanners miss vulnerabilities that only surface in agent orchestration, tool parameter construction, and inter-agent delegation. Context-aware testing driven by specialized attacker agents that adapt to what the application reveals finds the chained exploits that static scripts can't reach.

Desired outcome:

- You use context-aware, multi-agent attack simulation for penetration testing that adapts to the specific application under test.
- The testing system develops deep understanding of the application's architecture, data flows, and agent interactions, then executes sophisticated attack chains combining multiple vulnerability types.
- Findings are validated through actual exploitation, prioritized by real-world exploitability, and documented with reproducible attack paths and ready-to-implement fixes.

Common anti-patterns:

- Running generic vulnerability scanners against agentic AI systems without adapting test scenarios to the agent's specific capabilities and tool integrations, missing tool parameter injection, memory poisoning, and delegation-chain privilege escalation.
- Testing individual agent components in isolation without exercising multi-agent coordination paths, missing trust boundary violations and cascading failures from a compromised agent in an orchestration chain.
- Relying on predefined test scripts that don't adapt based on application responses, missing vulnerabilities that require dynamic exploration because agentic systems behave differently based on context and prior interactions.

Benefits of establishing this best practice:

- Context-aware testing adapts to the specific application, discovering vulnerabilities that static test scripts and generic scanners miss.
- Actual exploitation validates findings, reducing false positives and letting teams prioritize based on real risk.
- Specialized agents collaborate on reconnaissance, vulnerability analysis, exploit validation, and finding prioritization, identifying chained vulnerabilities that combine information disclosure with privilege escalation.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Penetration testing that chases agent vulnerabilities has to look like the attack it is simulating. Attacker agents don't run the same script on every target. They map the surface, probe for weaknesses, adapt to responses, and chain findings. Testing tools need to do the same or they miss exactly the scenarios that matter most for agentic systems.

A multi-agent penetration testing system orchestrates specialized security agents collaboratively. The system begins with baseline scanning to establish coverage, then conducts broad reconnaissance to map the application surface and identify initial attack vectors. Building on these findings, it dynamically generates focused test tasks tailored to the specific application context, reasoning about discovered endpoints, business logic patterns, and potential vulnerability chains.

[AWS Security Agent](#) provides on-demand penetration testing with this multi-agent approach. It deploys specialized AI agents that develop application context from provided documentation

and credentials, then execute attack chains to identify complex vulnerabilities conventional tools miss. The architecture includes agents for attack surface mapping, business logic analysis, finding validation, and vulnerability prioritization based on actual exploitability scored using the Common Vulnerability Scoring System (CVSS). The system performs chained attacks, combining an information disclosure flaw with privilege escalation to reach sensitive resources, or chaining insecure direct object references with authentication bypass, rather than stopping at single-vulnerability detection.

AWS Security Agent starts with the OWASP Top 10 and then customizes its approach based on the context it learns from documents and code. The agent adapts to the responses it receives, building a custom attack plan for each application. Provide target URLs, authentication details, source code, and documentation so the agent can develop deep application understanding before testing begins.

Agent-specific scenarios need manual supplementation. Prompt injection chains across agent boundaries, tool parameter manipulation, memory poisoning through crafted tool outputs, and human-in-the-loop bypass techniques all require scenarios that go beyond the OWASP baseline. Use the findings from AGENTSEC07-BP05 to inform the scenario library.

Implementation steps

1. **Provide application context to the testing agent:** Configure [AWS Security Agent](#) with target application details including URLs, authentication credentials (stored in AWS Secrets Manager), source code, and architecture documentation.
2. **Run tests across the full surface:** Execute on-demand penetration tests that exercise agent orchestration endpoints, tool invocation paths, and multi-agent communication channels.
3. **Triage validated findings by exploitability:** Review findings with reproducible attack paths, impact analysis, and suggested code fixes, and prioritize remediation based on CVSS scores and actual exploitability.
4. **Add agent-specific scenarios manually:** Supplement automated testing with scenarios targeting prompt injection chains, tool parameter manipulation, and multi-agent trust boundary violations.
5. **Track posture over time:** Store penetration test results and compare them across test cycles to measure security posture improvement.

Resources

Related best practices:

- [AGENTSEC07-BP05 Regular security assessments and red teaming](#)
- [AGENTSEC02-BP02 Validate tool inputs and outputs](#)
- [AGENTSEC09-BP01 Integrate AI-powered vulnerability scanning across the development lifecycle](#)

Related documents:

- [Inside AWS Security Agent: A multi-agent architecture for automated penetration testing](#)
- [AWS Security Agent FAQs](#)
- [Security Considerations for AWS Security Agent and AI assisted penetration testing](#)
- [OWASP Top 10 for LLM Applications](#)

Related services:

- [AWS Security Agent](#)
- [AWS Secrets Manager](#)
- [Amazon CloudWatch](#)

AGENTSEC09-BP03 Implement continuous security validation with automated remediation

Periodic assessments leave newly deployed agent capabilities exposed for weeks or months. On-demand validation integrated into the development pipeline, paired with automated fix suggestions, compresses the discovery-to-resolution loop from weeks to hours.

Desired outcome:

- You run security validation continually or on-demand as part of the development and deployment pipeline rather than only during periodic assessment windows.
- Validated findings arrive with ready-to-implement code fixes and configuration recommendations, so development teams remediate issues without waiting for security team intervention.

- You track remediation progress automatically, and regression testing confirms fixes are effective and don't introduce new vulnerabilities.

Common anti-patterns:

- Limiting penetration testing to annual or quarterly cycles, leaving newly deployed agent capabilities untested for long periods because agentic systems evolve rapidly with new tool integrations and capability expansions.
- Delivering vulnerability findings without practical remediation guidance, creating a bottleneck where development teams wait for security expertise to understand how to fix issues.
- Treating remediation as separate from discovery, losing context between the team that identified the issue and the team that must fix it and leading to incomplete fixes that address symptoms rather than root causes.

Benefits of establishing this best practice:

- On-demand testing validates security whenever new capabilities are deployed or significant changes are made, compressing the exposure window.
- Automated fix suggestions give development teams ready-to-implement code changes, closing the loop between discovery and resolution.
- Automated re-testing confirms fixes are effective and don't introduce new vulnerabilities.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The value of security validation is proportional to how fast it runs relative to how fast the application changes. Quarterly testing against a code base that changes weekly leaves most of the application untested most of the time, and agentic systems change faster than traditional applications because new tool integrations and capability expansions ship continuously. Integrating validation into CI/CD pipelines so it runs on every significant change keeps testing coverage aligned with application evolution.

Configure triggers for on-demand testing when new agent capabilities are added, tool integrations are modified, or permission boundaries are changed. [AWS Security Agent](#) transforms penetration testing from a weeks-long manual process into an on-demand capability that completes in hours. Each validated finding carries impact analysis, a reproducible attack path, and a ready-

to-implement code fix, which is what lets development teams remediate without waiting for specialized security expertise. Security teams define organizational requirements once and AWS Security Agent validates them during every design and code review, providing consistent enforcement at scale.

Remediation tracking monitors fix progress from discovery through resolution. Store test results and remediation status in a centralized system, and configure automated regression testing that re-runs relevant test scenarios after fixes are applied to confirm effectiveness. Amazon CloudWatch captures the security validation metrics that matter: time-to-detection, time-to-remediation, and fix effectiveness rates.

Agentic systems need validation triggers that traditional applications don't. Trigger security validation whenever agent system prompts are modified, new tools are registered, permission scopes are changed, or multi-agent orchestration patterns are updated. These changes introduce vulnerabilities that are not caught by standard code-level scanning because they affect agent behavior at the reasoning and orchestration layer.

Implementation steps

1. **Wire validation into CI/CD:** Integrate [AWS Security Agent](#) into CI/CD pipelines with triggers for on-demand security validation when agent capabilities, tool integrations, or permission boundaries change.
2. **Run code and design review on every PR:** Configure automated code and design security reviews that run on every pull request, giving developers real-time feedback during development.
3. **Route findings with fixes to owners:** Establish a remediation workflow that routes validated findings with suggested code fixes to the appropriate development team and tracks progress to resolution.
4. **Re-test after fixes automatically:** Implement regression testing that re-runs relevant security test scenarios after fixes are applied to confirm effectiveness.
5. **Measure and improve:** Monitor security validation metrics (time-to-detection, time-to-remediation, fix effectiveness) in Amazon CloudWatch and review trends to identify process improvements.

Resources

Related best practices:

- [AGENTSEC09-BP01 Integrate AI-powered vulnerability scanning across the development lifecycle](#)
- [AGENTSEC09-BP02 Conduct context-aware penetration testing with multi-agent attack simulation](#)
- [AGENTSEC07-BP05 Regular security assessments and red teaming](#)

Related documents:

- [AWS Security Agent](#)
- [AWS Security Agent FAQs](#)
- [Amazon Bedrock AgentCore adds quality evaluations and policy controls](#)

Related services:

- [Amazon CloudWatch](#)
- [AWS CodePipeline](#)

AGENTSEC09-BP04 Establish scoped and controlled testing environments for agent security assessments

Penetration testing that runs real exploit attempts against production agents can trigger tool calls, corrupt memory, or expose sensitive data. Scoped test environments with dedicated credentials and isolated agent state let teams run thorough assessments without risking production impact.

Desired outcome:

- You conduct security assessments for agentic AI systems in controlled environments with clearly defined scope, dedicated credentials, and isolation from production systems.
- You manage test credentials through secure vaulting services with automatic rotation, and testing activities are logged in your account for full auditability.
- The testing environment replicates production agent behavior closely enough to produce valid findings while containing the scope of exploit attempts.

Common anti-patterns:

- Running penetration tests directly against production agentic systems without scope controls, risking agents executing tool calls against production databases, sending messages to real users, or triggering downstream workflows in response to test inputs.
- Using shared or long-lived credentials for security testing, creating credential exposure risk where test credentials could be captured in logs, test artifacts, or finding reports.
- Not isolating test agent instances from production agent memory and state, letting test inputs and attack payloads pollute production agent memory and influence future agent behavior for real users.

Benefits of establishing this best practice:

- Environment isolation allows thorough security assessment, including real exploit attempts, without risking production impact or data corruption.
- Vaulting services limit credential exposure during testing and support automatic rotation after assessment cycles.
- Test logs stored in your account provide full visibility into testing activities for compliance and incident investigation.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

An agent under security test is an agent executing exploit attempts. In production, an agent running an exploit payload will actually call the tool it was told to call, send the email it was told to send, and write to the memory it was told to modify. The containment requirement is therefore structural: the test environment needs to look enough like production to produce valid findings, and it needs to be separated enough from production that exploit attempts can't reach real resources.

Provision dedicated testing environments that replicate the agent architecture, tool integrations, and data flows of production while maintaining isolation. Use separate agent instances with their own memory stores, tool endpoints, and permission boundaries. Configure test environments to use mock or sandboxed versions of external services where full production connectivity isn't required for valid testing results.

Manage test credentials through AWS Secrets Manager, storing static credentials (username and password) securely and supporting dynamic credential provisioning through AWS Lambda

functions for more complex authentication scenarios. [AWS Security Agent](#) supports both static credentials stored in AWS Secrets Manager and dynamic credentials accessed through AWS Lambda functions, giving flexible authentication that adapts to different application architectures. Rotate test credentials after each assessment cycle and audit credential access logs to detect any unauthorized usage.

Scope definition is a precondition, not an afterthought. Specify before each assessment which agent endpoints, tool integrations, and data stores are in scope and which are explicitly excluded. For AWS Security Agent penetration testing, provide target URLs, authentication details, source code, and documentation so the agent can develop application context within the defined scope. All test logs are stored in Amazon CloudWatch in your account for full visibility.

For multi-agent systems, test agent-to-agent communication paths in isolation before testing the full orchestration chain. That approach identifies trust boundary violations and delegation issues at the individual interaction level before they compound in complex multi-agent scenarios. Network segmentation and IAM policies help prevent test agent instances from reaching production resources.

Implementation steps

1. **Provision isolated test environments:** Replicate production agent architecture with isolated memory stores, tool endpoints, and permission boundaries.
2. **Manage credentials in Secrets Manager:** Store test credentials in AWS Secrets Manager with automatic rotation policies, or use AWS Lambda functions for dynamic credential provisioning for complex authentication scenarios.
3. **Define and document test scope:** Specify in-scope agent endpoints, tool integrations, and data stores for each assessment, with explicit exclusions for sensitive production resources.
4. **Contain the scope at the network and IAM layers:** Configure network segmentation and IAM policies that help prevent test agent instances from accessing production resources.
5. **Verify test logging in CloudWatch:** Confirm all test logs are captured in Amazon CloudWatch in your account, and review logs after each assessment to confirm scope adherence and identify unintended interactions.

Resources

Related best practices:

- [AGENTSEC09-BP02 Conduct context-aware penetration testing with multi-agent attack simulation](#)
- [AGENTSEC03-BP03 Implement least privilege with dynamic boundaries](#)
- [AGENTSEC07-BP04 Behavioral anomaly detection and agent containment](#)

Related documents:

- [Security Considerations for AWS Security Agent and AI assisted penetration testing](#)
- [AWS Security Agent FAQs](#)
- [AWS Secrets Manager documentation](#)
- [AWS Well-Architected Framework, Security Pillar](#)

Related services:

- [AWS Security Agent](#)
- [AWS Secrets Manager](#)
- [AWS Lambda](#)
- [Amazon CloudWatch](#)
- [AWS Identity and Access Management](#)

AGENTSEC09-BP05 Implement runtime threat detection, security event correlation, and automated remediation for agents

Scanning finds latent weaknesses. Runtime detection catches the attacks happening now. Correlating events across agent interaction surfaces and triggering automated remediation compresses the gap between an active exploit and the response that contains it.

Desired outcome:

- You continually monitor security events from agent activity, correlate them across interaction surfaces, and analyze them for multi-step attack sequences.
- You detect active threats targeting agentic systems within minutes, with critical attack sequences surfaced at the highest severity.
- Automated remediation workflows trigger containment actions and generate ready-to-implement fixes, reducing mean time to detection and mean time to remediation.

- Security teams have a unified view of agent-related threats alongside findings from vulnerability scanning and penetration testing.

Common anti-patterns:

- Treating agent security events in isolation rather than correlating them across interaction surfaces, missing multi-step attack sequences where individual events look benign but together constitute a coordinated attack.
- Relying on pre-deployment vulnerability scanning alone without runtime threat detection, leaving a gap where vulnerabilities introduced through configuration drift, new tool integrations, or novel attack techniques go undetected until the next scheduled assessment.
- Generating security alerts without automated remediation workflows, creating alert fatigue where security teams are overwhelmed by findings but lack the tooling to act quickly.
- Not correlating penetration testing findings with runtime threat detection signals, missing the connection between known vulnerabilities and active exploitation attempts that together provide a high-confidence remediation prioritization signal.

Benefits of establishing this best practice:

- AI/ML-powered event correlation identifies coordinated attacks spanning multiple agent interaction surfaces, time periods, and resources.
- Automated workflows trigger containment actions and generate fix recommendations when threats are detected, closing the loop between detection and response.
- Centralized findings from vulnerability scanning, penetration testing, and runtime threat detection enable risk-based prioritization across the full threat lifecycle.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Scanning and penetration testing find weaknesses before exploit. Runtime detection catches the exploit as it happens. Both are necessary because neither alone is sufficient. An agentic system adds surface area (tools, APIs, memory stores, other agents) that is exploited through prompt injection chains, credential misuse, data exfiltration sequences, and privilege escalation paths. Detection has to correlate across that surface, not treat each channel in isolation.

Deploy Amazon GuardDuty across all accounts where agents operate to provide continuous threat detection for agent IAM roles, API activity, and data access patterns. [GuardDuty Extended Threat Detection](#) correlates security signals to identify active attack sequences (privilege discovery followed by API manipulation, persistence activities, and data exfiltration) and surfaces them as critical-severity attack sequence findings with natural language summaries, MITRE ATT&CK mapping, and prescriptive remediation recommendations.

For agent-specific threat detection, configure GuardDuty monitoring across the data sources most relevant to agentic workloads:

- AWS CloudTrail management events for API call patterns
- Amazon VPC Flow Logs for network behavior
- DNS logs for command-and-control detection
- Amazon S3 data events for data access monitoring

Enable GuardDuty Runtime Monitoring for compute resources running agent workloads to detect threats at the operating system level, including suspicious process execution and network connections.

AWS Security Hub CSPM is the aggregation layer. Findings from Amazon GuardDuty (runtime threats), [AWS Security Agent](#) (vulnerability scanning and penetration testing), Amazon Macie (sensitive data exposure), and Amazon Inspector (software vulnerability scanning) normalize into the AWS Security Finding Format (ASFF) for consistent prioritization and automated response regardless of source. Security Hub CSPM insights correlate penetration testing findings with runtime detection signals, identifying cases where known vulnerabilities are being actively exploited (a high-confidence prioritization signal).

Amazon EventBridge rules trigger AWS Lambda functions or AWS Step Functions workflows when high-severity findings are generated. For agent-specific threats, the remediation workflow captures forensic state (agent memory, active sessions, recent tool invocations) to Amazon S3, applies containment actions (credential revocation, network isolation) as described in AGENTSEC07-BP04, generates remediation recommendations based on the finding type, and creates tracked remediation tasks. Findings from AWS Security Agent penetration testing that include suggested code fixes route directly to development teams through the existing remediation tracking workflow.

[Amazon Bedrock AgentCore Evaluations](#) adds a complementary detection layer. Continuous evaluation scoring detects behavioral drift that precedes or accompanies security incidents. A

sudden drop in safety or correctness scores combined with a GuardDuty finding for the same agent is a high-confidence signal that warrants immediate investigation. Amazon CloudWatch composite alarms triggered when both evaluation score degradation and a GuardDuty finding occur within the same time window surface those cases automatically.

Implementation steps

1. **Enable GuardDuty across agent accounts:** Turn on Amazon GuardDuty with monitoring configured for AWS CloudTrail events, Amazon VPC Flow Logs, DNS logs, Amazon S3 data events, and GuardDuty Runtime Monitoring for agent compute resources.
2. **Centralize findings in Security Hub CSPM:** Aggregate findings from Amazon GuardDuty, [AWS Security Agent](#), Amazon Macie, and Amazon Inspector in AWS Security Hub CSPM, and configure Security Hub CSPM insights to correlate penetration testing findings with runtime threat detection signals.
3. **Automate containment on high-severity findings:** Use Amazon EventBridge rules and AWS Lambda functions to trigger containment actions and generate fix recommendations when high-severity findings are generated.
4. **Combine evaluation drift with GuardDuty findings:** Configure Amazon CloudWatch composite alarms that combine [Amazon Bedrock AgentCore Evaluations](#) score degradation with GuardDuty findings to surface high-confidence threat signals.
5. **Route fixes to developers and measure MTTR:** Route remediation recommendations, including code fixes from AWS Security Agent, to development teams through a tracked workflow, and monitor mean time to detection and mean time to remediation as key security metrics.
6. **Tune detection quarterly:** Review detection rules, remediation workflows, and finding correlation logic quarterly based on observed threat patterns and false positive rates.

Resources

Related best practices:

- [AGENTSEC05-BP01 Implement comprehensive logging and decision artifact storage](#)
- [AGENTSEC07-BP04 Behavioral anomaly detection and agent containment](#)
- [AGENTSEC09-BP01 Integrate AI-powered vulnerability scanning across the development lifecycle](#)
- [AGENTSEC09-BP02 Conduct context-aware penetration testing with multi-agent attack simulation](#)

Related documents:

- [Amazon GuardDuty Extended Threat Detection](#)
- [Amazon GuardDuty documentation](#)
- [AWS Security Hub CSPM documentation](#)
- [Automate cloud security vulnerability assessment and alerting using Amazon Bedrock](#)
- [How government agencies can transform cybersecurity operations with Amazon Bedrock AgentCore](#)
- [AWS Security Agent](#)

Related services:

- [Amazon GuardDuty](#)
- [AWS Security Hub CSPM](#)
- [Amazon EventBridge](#)
- [AWS Lambda](#)
- [AWS Step Functions](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [Amazon Macie](#)
- [Amazon Inspector](#)

Reliability

When agents reason, plan, and act through large language models, reliability takes on dimensions that traditional infrastructure patterns don't address. LLM decisions are stochastic, multi-agent coordination introduces new failure modes, and memory integrity becomes a first-class concern. An agent that works perfectly in testing may behave unpredictably in production when context windows fill up, models return unexpected outputs, or downstream agents become unavailable. This pillar provides best practices for building agent systems that execute tasks predictably, recover from failures automatically, and maintain partial functionality even under adverse conditions.

Capabilities

- [Predictable agent behavior](#)
- [Predictable task execution](#)
- [Agent memory and state management](#)
- [Multi-agent orchestration](#)
- [Agent cognition](#)
- [Legacy system integration](#)
- [Agent monitoring, management and recovery](#)
- [Graceful degradation and configuration management](#)

Design principles

In addition to the lens-level design principles, the reliability best practices in this lens are represented by at least one of the following principles:

- **Decouple agents through durable messaging:** Persistence, retry, and dead-letter handling absorb transient failures inside the messaging layer instead of cascading them through synchronous call chains.
- **Constrain blast radius through atomic responsibilities:** Single-task agents with the minimum permissions and clear instructions limit how far any individual failure or misbehavior can propagate.
- **Recover from the last known good state, not the beginning:** Checkpointed workflows, idempotent steps, and graceful degradation let work resume after a fault rather than restart from scratch.

- **Make multi-agent coordination resilient by design:** Arbiter patterns, capability taxonomies, and fallback paths keep collaborative workflows running when individual agents become unavailable or unreliable.
- **Ground reasoning in verifiable evidence:** Retrieval from authoritative sources, explicit citation, and hallucination detection keep agent outputs traceable to real data instead of fabricated content.
- **Exercise failure paths regularly:** Inject faults, run degraded-dependency tests, and rehearse recovery procedures. The first time a failure mode occurs should not be in production.

Predictable agent behavior

Agents combine compute, memory, cognition, and orchestration components, and each component is a point where a workflow can fail. Organizations that run agents on resilient messaging, modular fault isolation, and adaptive provisioning spend less time repairing infrastructure and more time improving agent capabilities.

AGENTREL01: How do I develop reliable agentic systems?

Capability intent

- Agent-to-agent and agent-to-system communication runs through a durable messaging substrate, so transient failures are absorbed through persistence, retry, and dead-letter handling rather than cascading across the workflow.
- Compute, memory, cognition, and orchestration operate as independent layers with well-defined contracts, so a surge or failure in one layer stays contained and the remaining layers keep operating in a known degraded mode.
- Specialized agents each own a single capability, their own state, and a narrow permission scope, so failures are isolated to the agent that encountered them and the broader environment continues running.
- Inter-agent communication follows a consistent taxonomy of message schemas, versioned endpoints, error formats, and retry policies, so agents compose into workflows without custom translation layers between every pair.

- Compute, inference, and model-tier allocation adapt in real time to workload and capacity signals, so agents maintain steady performance under variable load without manual capacity planning.

Maturity levels

These levels summarize what each stage of maturity looks like for predictable agent behavior as a whole.

Level	Name	What it looks like
1	Initial	Agents call each other directly over synchronous channels with ad-hoc message formats and no durable queue between them. A single slow or failing agent cascades into the rest of the workflow, and there is no consistent way to trace which hop lost a message. Capacity is provisioned statically, so demand spikes cause throttling and low-demand periods waste resources.
2	Emerging	Teams have introduced a messaging layer (typically Amazon SQS and Amazon EventBridge) for the most critical agent-to-agent paths, and dead-letter queues catch repeated failures. Agents run on Amazon Bedrock AgentCore Runtime with basic layer separation, and on-demand Amazon

Level	Name	What it looks like
		Bedrock inference absorbs most capacity variability. Communication schemas are documented but not uniformly enforced.
3	Defined	Durable messaging is the default for agent-to-agent and agent-to-system communication, with schemas registered in EventBridge Schema Registry and workflows orchestrated through AWS Step Functions . Specialized agents run as single-purpose actors on Amazon Bedrock AgentCore Runtime with scoped IAM roles, communicate through Amazon Bedrock AgentCore Gateway or the A2A protocol , and expose per-agent metrics through Amazon Bedrock AgentCore Observability . Tiered model selection routes simple tasks to smaller models and reasoning-heavy work to larger ones.

Level	Name	What it looks like
4	Proactive	Fail-fast logic, fallback behaviors, and runtime capability toggling are enforced automatically at layer boundaries, and Amazon Bedrock AgentCore Policy enforces standardized access control at the gateway boundary through Cedar policies. Cross-Region inference is the default for production agents, and Bedrock Provisioned Throughput underwrites latency-sensitive paths. Contract tests in CI/CD help prevent protocol regressions, and scheduled scaling pre-provisions capacity ahead of known demand patterns.

Level	Name	What it looks like
5	Optimized	The messaging substrate , layer isolation, inter-agent contracts, and capacity allocation are continuously recalibrated from observability data rather than revised on a fixed review cadence. New workflows inherit the resilience pattern by default through reusable templates and shared services, and the organization contributes its agent reliability patterns (messaging topology, fail-fast envelopes, tiered model routing) back to its communities of practice.

Common issues to watch for

- Teams wire up agent-to-agent communication over direct synchronous calls and only discover the coupling when one agent's latency or failure takes the whole workflow down.
- Agent architectures ship as a single service rather than as separate compute, memory, cognition, and orchestration layers, so an issue in any component forces a full restart instead of a contained fix.
- Specialized agents accrete unrelated tools and broader system prompts over time, expanding their failure radius and making issues harder to reproduce as responsibilities blur.
- Message schemas, error formats, and retry policies are defined per agent pair rather than across the agent fleet, so every new agent introduces its own translation layer and every change risks breaking a downstream consumer.
- Capacity is provisioned for the worst case rather than adapted to demand. Peak traffic still produces throttling despite idle headroom elsewhere, and low-demand periods pay for unused capacity.

Best practices

- [AGENTREL01-BP01 Implement a resilient messaging layer](#)
- [AGENTREL01-BP02 Establish modular, fault-isolated layers](#)
- [AGENTREL01-BP03 Design specialized agents following actor model principles](#)
- [AGENTREL01-BP04 Standardize communication protocols](#)
- [AGENTREL01-BP05 Implement adaptive provisioning](#)

AGENTREL01-BP01 Implement a resilient messaging layer

Direct agent-to-agent calls couple failure modes. When one agent fails, everything downstream fails with it. A messaging layer with persistence, retry, and dead-letter handling absorbs transient faults and lets workflows resume from where they stopped.

Desired outcome:

- Your agents communicate through an intermediary messaging layer with persistence, retry, and dead-letter handling rather than direct synchronous calls.
- You have durable workflow state that survives the restart or loss of any single component.
- You can trace every agent message across synchronous and asynchronous boundaries.

Common anti-patterns:

- Wiring agents together through direct synchronous calls, so a single failure cascades through every dependent agent.
- Running messaging infrastructure without persistence, making workflow recovery impossible after a component outage.
- Treating every interaction as synchronous, creating bottlenecks that block independent agent operation.

Benefits of establishing this best practice:

- Persistence and retry contain transient failures within the messaging layer instead of exposing them as agent outages.
- Dead-letter handling helps prevent poison messages from blocking healthy workflow execution.

- A durable messaging substrate is the foundation for advanced orchestration patterns including saga and arbiter.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Every agent-to-agent call is a coupling decision. Synchronous calls tie the caller's availability to the callee's availability. In a network of agents that multiplies quickly. Five agents with four synchronous dependencies, and the availability product drops below any single agent's SLA. A messaging layer breaks the coupling by buffering the call in durable infrastructure. The caller emits a message and moves on. The receiver processes it on its own schedule, with retries and dead-letter routing handled outside the agent's own code.

Pattern selection follows the interaction shape. Use [Amazon EventBridge](#) for content-based routing where a single event fans out to multiple consumers, with EventBridge Schema Registry documenting the contract between agents. Use [Amazon SQS](#) for durable point-to-point delivery with configurable visibility timeouts and dead-letter queues. Use Amazon SNS for fan-out to multiple downstream consumers.

Workflow durability ties the messaging layer to business outcomes. A message that reaches its queue still needs orchestration to coordinate multi-step work across agents. [AWS Step Functions](#) persists execution state at every step transition, so recovery starts from the last completed step rather than the beginning. Without that persistence, a failure in step five of a seven-step workflow re-executes every prior step, wasting compute and risking duplicate side effects. Dead-letter handling complements durability. Poison messages get isolated for triage rather than blocking healthy traffic behind them.

Implementation steps

1. **Map every agent communication path and classify it:** Document each interaction as synchronous direct communication (A2A), loosely coupled tool invocation (MCP), or asynchronous event-driven through [Amazon EventBridge](#).
2. **Configure EventBridge rules and SQS queues:** Set up Amazon EventBridge content-based routing for event-driven paths and [Amazon SQS](#) queues for durable point-to-point messaging.
3. **Define event schemas in EventBridge Schema Registry:** Register a schema for each agent message type so sender and receiver agree on the contract.

4. **Configure dead-letter queues with automated triage:** Route repeatedly failed messages to DLQs and wire Amazon CloudWatch alarms so operators see poison messages before they block traffic.
5. **Instrument the messaging layer with AgentCore Observability:** Enable [Amazon Bedrock AgentCore Observability](#) for distributed tracing so you can follow a message across EventBridge, SQS, and agent boundaries.

Resources

Related best practices:

- [AGENTREL01-BP02 Establish modular, fault-isolated layers](#)
- [AGENTREL01-BP03 Design specialized agents following actor model principles](#)
- [AGENTREL01-BP04 Standardize communication protocols](#)

Related documents:

- [Amazon EventBridge](#)
- [Amazon SQS](#)
- [AWS Step Functions](#)
- [Build resilient generative AI agents](#)
- [Operationalizing agentic AI on AWS](#)

Related services:

- [Amazon EventBridge](#)
- [Amazon SQS](#)
- [Amazon SNS](#)
- [AWS Step Functions](#)
- [Amazon Bedrock AgentCore](#)

AGENTREL01-BP02 Establish modular, fault-isolated layers

Monolithic agent architectures force every fault to become a full-system fault. Splitting compute, memory, reasoning, and orchestration into independently scalable layers with fail-fast boundaries keeps the scope of impact small. Teams keep serving requests at reduced capability instead of going unavailable.

Desired outcome:

- Your agent stack is split into distinct layers (compute, memory, reasoning, orchestration, and tool integration) with documented API contracts at each boundary.
- You have fail-fast behavior on inter-layer calls, with defined fallback modes for each degraded state.
- You can toggle non-critical capabilities at runtime without redeploying.

Common anti-patterns:

- Deploying monolithic agents where a failure in any component forces a full restart for issues that should be isolated.
- Running without automatic cutoffs, allowing latency or error rates in one component to propagate through every dependent call.
- Treating all capabilities as equally critical, missing the chance to keep core functionality available when non-essential components fail.

Benefits of establishing this best practice:

- Teams can develop, test, and deploy individual layers independently without blocking on the rest of the stack.
- Fault isolation narrows troubleshooting to the layer that actually failed rather than the whole system.
- Graceful degradation keeps agents responsive even when individual layers are unavailable.

Level of risk exposed if this best practice is not established: High

Implementation guidance

AgentCore Runtime organizes capabilities into distinct layers (Runtime, Memory, Gateway, and Identity), each addressable independently. Design your agents to treat these as separate failure domains. If Memory becomes unavailable, your agent's routing logic (Gateway) and authentication (Identity) should continue functioning. Implement health checks per layer and configure independent timeout and retry policies for each, rather than treating AgentCore as a monolithic dependency.

When a downstream layer's error rate climbs, the caller should stop waiting for timeouts and activate a fallback. Examples include session-only context instead of long-term memory, an alternative Amazon Bedrock model instead of the primary, or a cached answer instead of fresh retrieval. Without fail-fast, every degraded call consumes thread budget and propagates latency back to the user. The [AWS fail-fast pattern guidance](#) covers the mechanics.

Runtime capability toggling keeps the scope of impact small during an incident. If one tool is flaky, turn that tool off and keep the rest of the agent operational rather than taking the whole agent down. Publish structured health status per layer through [Amazon Bedrock AgentCore Observability](#) so downstream components adapt to the toggle state automatically. Service maps give operators the view they need to correlate layer health to user-visible symptoms.

Implementation steps

- 1. Decompose the architecture into layers with documented contracts:** Split the agent into distinct layers for compute, memory, cognition, orchestration, and tool integration. Publish the API contract at every boundary.
- 2. Deploy each layer independently on AgentCore Runtime:** Run each layer on [Amazon Bedrock AgentCore Runtime](#) with no shared execution resources between layers.
- 3. Implement fail-fast logic per inter-layer call:** For each call boundary, define the error-rate threshold that trips the cutoff and the fallback behavior that takes over.
- 4. Publish structured layer health through AgentCore Observability:** Emit per-layer health signals through [Amazon Bedrock AgentCore Observability](#) so downstream components can adapt and operators can trace degradation to its source.
- 5. Wire runtime capability toggling:** Build a control plane that disables non-critical capabilities without redeployment so operators can contain incidents as they happen.

Resources

Related best practices:

- [AGENTREL01-BP01 Implement a resilient messaging layer](#)
- [AGENTREL01-BP03 Design specialized agents following actor model principles](#)
- [AGENTREL08-BP01 Establish consistent configuration management practices](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Build resilient generative AI agents](#)
- [Operationalizing agentic AI on AWS](#)
- [AWS fail-fast pattern](#)

Related videos:

- [AWS re:Invent 2024 - Architecting scalable and secure agentic AI with AgentCore \(AIM431\)](#)
- [AWS re:Invent 2024 - Balance cost, performance & reliability for AI at enterprise scale \(AIM3304\)](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTREL01-BP03 Design specialized agents following actor model principles

Multi-purpose agents concentrate risk. A single failure in one function affects every other function in the same process. Specialized agents that encapsulate one atomic capability, own their state, and communicate through messages keep the impact proportional to the scope of the failing component.

Desired outcome:

- You have each agent scoped to a single atomic function with explicit input and output schemas.
- Your agents maintain their own state and exchange information through explicit message passing, not shared memory.
- You can scale, replace, or upgrade any one agent without disturbing the others.

Common anti-patterns:

- Building multi-purpose agents that combine disparate functions in one component, reducing the ability for independent scaling and widening the failure impact.
- Coupling agents through shared in-memory state rather than explicit message passing, so a crash in one process corrupts another.
- Designing agents as general-purpose processors with overlapping capabilities, making issue reproduction and remediation harder than it needs to be.

Benefits of establishing this best practice:

- A failure stays contained to the agent that encountered it, preserving overall workflow integrity.
- Single-responsibility agents are simpler to test deterministically because the input-output contract is small.
- Each agent scales, deploys, and is replaced on its own schedule, which keeps the rest of the system stable during change.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The actor model is a discipline, not a runtime feature. A managed runtime can give you isolated execution. [Amazon Bedrock AgentCore Runtime](#) runs each agent in its own microVM with dedicated tool permissions. But whether your agents actually follow actor principles depends on how you scope the system prompt, how narrowly you define the tool set, and whether inter-agent calls go through messages or shared state.

Give each agent a single, well-scoped system prompt that constrains it to one domain. Use Strands Agents or another agentic framework, but the test is framework-independent. Can you describe the agent's job in a single sentence without using "and"? If not, the agent carries more than one responsibility and should be split.

Communication pattern follows interaction shape. Use [Amazon Bedrock AgentCore Gateway](#) to expose agents as discoverable MCP tools when the caller treats the specialized agent as a capability it invokes on demand. Use the [Agent-to-Agent \(A2A\) protocol](#) when agents need peer collaboration with streaming responses or multi-turn exchanges. Monitor each agent individually. Track task success rate, processing latency, and error rate through [Amazon Bedrock AgentCore Observability](#) so one agent's issues don't hide under aggregate fleet metrics.

Implementation steps

1. **Decompose the workflow into specialized agents:** Define each agent around a single atomic function with explicit input and output schemas.
2. **Deploy each agent on AgentCore Runtime with a dedicated IAM role:** Run each agent on [Amazon Bedrock AgentCore Runtime](#) with an execution role scoped to only the resources required for that agent's function.
3. **Choose the inter-agent communication pattern:** Use [Amazon Bedrock AgentCore Gateway](#) for loosely coupled tool-style invocation, or the [A2A protocol](#) for peer-to-peer collaboration with richer interaction patterns.
4. **Monitor each agent independently:** Configure [Amazon Bedrock AgentCore Observability](#) with per-agent metrics and alarms so one agent's issues are not masked by fleet-level aggregates.

Resources

Related best practices:

- [AGENTREL01-BP01 Implement a resilient messaging layer](#)
- [AGENTREL01-BP02 Establish modular, fault-isolated layers](#)
- [AGENTREL01-BP04 Standardize communication protocols](#)
- [AGENTSUS01-BP01 Design specialized agents with explicit resource boundaries](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Deploy A2A servers in AgentCore Runtime](#)
- [Introducing Amazon Bedrock AgentCore Gateway](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)

AGENTREL01-BP04 Standardize communication protocols

Custom message formats between every agent pair turn new integrations into one-off engineering projects. Standardized schemas, versioned endpoints, and a canonical error format let agents compose into workflows without a translation layer at every boundary.

Desired outcome:

- You have a canonical message schema, error format, and retry policy that every agent follows.
- You version endpoints and maintain backward compatibility so existing integrations keep working when protocols evolve.
- You enforce protocol adherence through automated contract tests in the CI/CD pipeline.

Common anti-patterns:

- Building ad-hoc communication patterns with custom message formats per interaction, producing translation layers between every agent pair.
- Evolving endpoints without versioning or backward compatibility, breaking existing integrations on each change.
- Allowing each agent to set its own timeout, retry logic, and error response format, producing unpredictable failure behavior across the fleet.

Benefits of establishing this best practice:

- Consistent schemas and contracts reduce integration complexity and remove point-to-point translation code.
- Predictable multi-agent orchestration becomes possible because agents compose into workflows without hardcoded dependencies.

- New agents can be introduced or replaced without rewriting dependent components.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

[Amazon Bedrock AgentCore Gateway](#) provides a managed layer for agent discovery and tool invocation with built-in authentication and authorization. Underneath it, the Agent-to-Agent (A2A) protocol standardizes direct agent-to-agent communication and the Model Context Protocol (MCP) standardizes agent-to-tool interactions. Choosing these protocols instead of inventing your own pays off every time a new agent joins the network.

If every agent invents its own error codes and retry guidance, a caller can't write a single error-handling path. A canonical format with three fields (error code, correlation ID, and retry guidance) covers nearly every case and lets callers apply the same logic regardless of which agent returned the error. [Amazon Bedrock AgentCore Policy](#) enforces who can call what at the gateway boundary through Cedar policies, so the contract is enforced before the request reaches the agent rather than relying on documentation alone.

Versioning matters because protocols evolve. Version every AgentCore Gateway target so callers can migrate at their own pace. Register message schemas for each agent interaction type so serialization is consistent across boundaries. Wire contract tests into CI/CD so protocol regressions get caught before deployment rather than during an incident.

Implementation steps

1. **Define the canonical communication taxonomy:** Document the standard message schemas, error response format, and retry policies that every agent follows.
2. **Configure AgentCore Gateway with A2A and MCP protocols:** Use [Amazon Bedrock AgentCore Gateway](#) as the managed surface for standardized agent-to-agent and agent-to-tool communication.
3. **Enforce access control with AgentCore Policy:** Apply Cedar policies through [Amazon Bedrock AgentCore Policy](#) so the gateway rejects unauthorized calls at the boundary.
4. **Implement canonical error handling across all agent interfaces:** Propagate a correlation ID through every call and return errors in the canonical format so callers can handle them uniformly.
5. **Run automated contract tests in CI/CD:** Block deployment when a protocol regression is detected so protocol standards stay enforced as the agent fleet grows.

Resources

Related best practices:

- [AGENTREL01-BP01 Implement a resilient messaging layer](#)
- [AGENTREL01-BP03 Design specialized agents following actor model principles](#)
- [AGENTREL01-BP05 Implement adaptive provisioning](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Policy](#)
- [Transform your MCP architecture: Unite MCP servers through AgentCore Gateway](#)
- [Secure AI agents with Policy in Amazon Bedrock AgentCore](#)
- [Strands Agents A2A Protocol](#)
- [Strands Agents MCP Tools](#)

Related videos:

- [Integrating MCP Tools with Strands Agents](#)
- [Breaking multi-agent silos: A2A + MCP in action with Strands Agents](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)

AGENTREL01-BP05 Implement adaptive provisioning

Static provisioning forces a choice between overpaying for peak and failing under load. Agent workloads shift minute to minute, so capacity, model tier, and quota must respond to task complexity and current demand without operator intervention.

Desired outcome:

- You have agent compute allocation that adjusts for each invocation without manual tuning.
- You route tasks to model tiers appropriate for their complexity, with fallbacks when quotas tighten.
- You pre-provision resources ahead of known demand patterns and scale down during quiet periods.

Common anti-patterns:

- Running static resource provisioning and paying for peak capacity even during low-demand periods.
- Skipping the metrics that trigger scaling decisions, so the system has no basis to provision resources when they are needed.
- Treating every task as needing the same model, ignoring the cost and latency savings of tiering by complexity.

Benefits of establishing this best practice:

- Performance stays consistent under variable load because resources track demand instead of a fixed provisioning plan.
- Resource exhaustion during spikes is prevented without human intervention.
- Cost drops during low-demand periods through automatic scale-down.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Serverless is the baseline for adaptive compute. [Amazon Bedrock AgentCore Runtime](#) hosts agents with built-in scaling that adjusts compute allocation for each invocation, so individual agents don't need to manage fleet sizing. For LLM inference, Amazon Bedrock's on-demand mode scales without capacity reservations. [Amazon Bedrock cross-region inference](#) distributes requests across Regions to reduce the impact of regional capacity constraints.

Tiering matches model selection to workload. Low-complexity tasks route to smaller, faster Amazon Bedrock models, while complex reasoning routes to larger models. The router should

adjust dynamically based on task complexity signals and current quota utilization, not a fixed rule baked into code. For latency-sensitive user-facing agents where throttling is unacceptable, use Amazon Bedrock's Priority on-demand tier for premium throughput allocation. For workloads that need consistent low latency regardless of overall service demand, use Amazon Bedrock Provisioned Throughput with fixed model units. The [Amazon Bedrock Capacity, Limits, and Cost Optimization](#) guide covers the trade-offs between Flex, Standard, Priority, and Reserved tiers.

Monitor composite health signals across agent layers and trigger coordinated scaling actions when the system approaches capacity limits. Token throughput, model-level latency, and error rates for each layer through [Amazon Bedrock AgentCore Observability](#) are the signals that drive tier adjustments. Scheduled scaling handles anticipated demand. If historical data shows a spike every Monday at 9 a.m., pre-provision before the spike lands rather than reacting during it.

Implementation steps

- 1. Deploy agents on AgentCore Runtime for serverless scaling:** Use [Amazon Bedrock AgentCore Runtime](#) so compute allocation adjusts for each invocation without manual fleet sizing.
- 2. Route tasks by complexity to appropriate Amazon Bedrock models:** Implement tiered model selection that sends low-complexity tasks to smaller models and reasoning-heavy tasks to larger ones based on complexity signals.
- 3. Enable Amazon Bedrock cross-region inference:** Turn on [cross-region inference](#) to distribute requests and reduce the impact of regional capacity constraints.
- 4. Monitor token throughput and latency through AgentCore Observability:** Watch per-tier throughput and latency through [Amazon Bedrock AgentCore Observability](#) and trigger tier adjustments when thresholds are exceeded.
- 5. Use scheduled scaling ahead of anticipated spikes:** Pre-provision based on historical patterns so capacity is ready before demand lands.

Resources

Related best practices:

- [AGENTREL01-BP01 Implement a resilient messaging layer](#)
- [AGENTREL01-BP02 Establish modular, fault-isolated layers](#)
- [AGENTREL08-BP03 Architect agent systems with resource isolation and contention mitigation](#)
- [AGENCOST02-BP01 Architect tiered model selection for cost-performance optimization](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Securely launch and scale your agents and tools on Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock cross-region inference](#)
- [Amazon Bedrock Capacity, Limits, and Cost Optimization](#)
- [Effective cost optimization strategies for Amazon Bedrock](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Runtime](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Predictable task execution

Agents that constrain LLM stochasticity through atomic task design, least-privilege permissions, and clear instruction protocols deliver predictable outcomes even when the underlying models are non-deterministic. Agent reliability extends beyond supporting infrastructure to the reliability of executing the intended task with the appropriate data at the correct time.

AGENTREL02: How do you develop agentic systems that reliably execute tasks with predictable outcomes?

Capability intent

- Each agent owns a single atomic capability with explicit input validation and a structured output schema, so LLM stochasticity is bounded by narrow, testable contracts.

- Every agent operates within a least-privilege permission envelope enforced at identity, policy, and access-control layers, so an unexpected model decision affects only the systems explicitly authorized for that agent.
- Agents emit agent-specific telemetry (prompts, tool calls, memory access, output quality) that is compared against behavioral baselines, so drift and anomalies are detected before they cascade into failures.
- Instructions reach agents through canonical prompt templates, versioned configuration, and explicit handoff schemas, so interpretation of objectives is consistent across single-agent and multi-agent workflows.
- Agent actions are routed to the appropriate tier of human oversight based on risk and reversibility, so high-consequence decisions receive review without adding latency to routine work.

Maturity levels

These levels summarize what each stage of maturity looks like for predictable task execution as a whole.

Level	Name	What it looks like
1	Initial	Agents are general-purpose processors with broad system prompts and ambiguous input and output contracts . Permissions are coarse-grained, logging is generic and lacks agent-specific decision points, prompts are ad-hoc and unversioned, and every agent action receives the same level of human review, or none at all.
2	Emerging	Teams have started decomposing workflows into single-purpose agents and defining input and output

Level	Name	What it looks like
		schemas. Each agent has a dedicated IAM execution role, Amazon Bedrock AgentCore Observability captures per-agent telemetry, and prompt templates live in shared documentation. Some high-risk actions require human approval, though classification is informal.
3	Defined	Atomic agents run on Amazon Bedrock AgentCore Runtime with structured output enforcement and regular validation through Amazon Bedrock AgentCore Evaluations . Access is restricted through Amazon Bedrock AgentCore Identity and AWS Identity and Access Management (IAM) policies scoped per agent. Behavioral baselines drive alerts through Amazon CloudWatch Anomaly Detection , prompt templates are versioned, and a documented risk framework routes agent actions into autonomous, notify, and approve tiers.

Level	Name	What it looks like
4	Proactive	Access boundaries are enforced through Amazon Bedrock AgentCore Policy with Cedar policies at the gateway, and Amazon Bedrock Guardrails intercept policy-violating outputs before escalation to human reviewers. Prompt-version comparisons run through Amazon Bedrock AgentCore Evaluations before traffic migrates, IAM Access Analyzer drives least-privilege remediations from AWS CloudTrail data, and approval workflows carry timeouts, escalation paths, and full audit context.

Level	Name	What it looks like
5	Optimized	Atomic task contracts, least-privilege scopes, anomaly baselines, prompt libraries , and oversight tiers are continuously recalibrated from observability data. Automated responses quarantine anomalous agents, adversarial contract tests block prompt-injection regressions in CI/CD, and the organization publishes its agent reliability patterns and measurements back to its communities of practice.

Common issues to watch for

- Agents accumulate broad, overlapping responsibilities over time, so a single misinterpretation can affect multiple capabilities and failure modes become harder to reproduce as scope expands.
- IAM execution roles and policy boundaries are written with wildcards or at the convenience of a first deployment, so the scope of impact of any unpredicted LLM action is wider than the agent's legitimate function.
- Monitoring captures infrastructure signals but not agent-specific decision points, so behavioral drift (longer outputs, more tool calls, or shifts in output distribution) is invisible until it produces a user-visible failure.
- Prompts and handoff formats are authored ad-hoc by each team, so agents interpret objectives inconsistently and multi-agent workflows break when either side of a handoff evolves independently.
- All agent actions receive the same level of human review, either uniform approval that bottlenecks automation or uniform autonomy that lets high-consequence decisions ship without oversight.

Best practices

- [AGENTREL02-BP01 Design agents for specific and atomic tasks](#)
- [AGENTREL02-BP02 Limit agent permissions to minimum required access](#)
- [AGENTREL02-BP03 Implement behavioral anomaly detection and monitoring](#)
- [AGENTREL02-BP04 Develop clear instruction protocols for agents](#)
- [AGENTREL02-BP05 Establish tiered human oversight and approval workflows](#)

AGENTREL02-BP01 Design agents for specific and atomic tasks

LLM outputs are stochastic, so the scope of an agent's responsibility is also the scope of that stochasticity. Narrow, atomic agents with explicit input contracts and structured output schemas make the unpredictable parts of the model more observable, testable, and containable.

Desired outcome:

- You have each agent scoped to one atomic function with explicit input validation and a structured output schema.
- Your agents are tested with deterministic, representative inputs before deployment.
- You track per-agent quality metrics (schema compliance rate and task completion rate) so regressions appear before users see them.

Common anti-patterns:

- Building multi-purpose agents that combine disparate functions, widening the surface area for unpredictable model behavior.
- Skipping explicit input contracts and output schemas, allowing ambiguous data flows that amplify LLM stochasticity.
- Treating agents as general-purpose processors without discrete responsibilities, making issue reproduction hard.

Benefits of establishing this best practice:

- Constrained scope shrinks the surface for unpredictable behavior.
- Composable units can be independently validated with deterministic test cases.

- Clear operational boundaries speed up issue identification and remediation.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Atomic design starts with the prompt. A single, well-defined system prompt that describes exactly one capability is the easiest constraint to enforce and the hardest to violate silently. Pair it with explicit input validation that rejects out-of-scope requests before the LLM is invoked, and a structured output schema that constrains the response format. Amazon Bedrock's tool use (function calling) limits the model's action space to operations relevant to the agent's function. Amazon Bedrock's [structured output](#) feature enforces JSON schema compliance at the model level to reduce output validation failures.

Deployment reinforces the atomic boundary. Each agent runs on [Amazon Bedrock AgentCore Runtime](#) with its own execution context and dedicated tool permissions, so a misconfigured or compromised agent can't reach beyond its scope. This gives you a physical boundary that matches the logical one you defined in the prompt.

Testing is where atomic design pays off. Because each agent has a narrow contract, you can run [Amazon Bedrock AgentCore Evaluations](#) against representative inputs and compare outputs to expected results. LLM outputs are not fully deterministic even with temperature=0, so the tests should validate semantic correctness rather than exact string matching. Monitoring each agent through [Amazon Bedrock AgentCore Observability](#) tracks schema compliance rate and task completion rate, with alarms when metrics drift from baselines.

Implementation steps

1. **Decompose workflows into atomic agents:** Give each agent a single system prompt, a constrained tool set, and explicit input/output schemas.
2. **Deploy on AgentCore Runtime with structured output enforcement:** Run each agent on [Amazon Bedrock AgentCore Runtime](#) with dedicated tool permissions and [Bedrock structured output](#) enforcement.
3. **Implement input validation that rejects out-of-scope requests:** Validate inputs before the LLM is called so malformed or out-of-scope requests don't reach the model.
4. **Validate behavior with AgentCore Evaluations:** Use [Amazon Bedrock AgentCore Evaluations](#) with representative inputs and golden-path test cases, asserting semantic correctness rather than exact strings.

5. **Monitor per-agent quality metrics:** Track schema compliance rate and task completion rate through [Amazon Bedrock AgentCore Observability](#) with alarms for baseline deviations.

Resources

Related best practices:

- [AGENTRELO1-BP03 Design specialized agents following actor model principles](#)
- [AGENTRELO2-BP02 Limit agent permissions to minimum required access](#)
- [AGENTRELO2-BP03 Implement behavioral anomaly detection and monitoring](#)
- [AGENTRELO2-BP04 Develop clear instruction protocols for agents](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Evaluations](#)
- [Build reliable AI agents with Amazon Bedrock AgentCore Evaluations](#)
- [Amazon Bedrock structured output](#)
- [Strands Agents Custom Tools](#)

Related videos:

- [Strands Tools: Building Custom AI Agents with Python](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Evaluations tutorials](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)

AGENTREL02-BP02 Limit agent permissions to minimum required access

Broad permissions turn a misinterpreted instruction into a cascading incident. Least-privilege access keeps the scope of impact of unpredictable LLM behavior narrow and makes anomalous activity more visible against a well-defined baseline.

Desired outcome:

- You have each agent granted only the permissions required for its specific function.
- You apply runtime access boundaries at the gateway, so the LLM's reasoning can't widen the agent's reach.
- You audit agent policies continually and remove permissions that actual usage doesn't justify.

Common anti-patterns:

- Granting broad permissions beyond the agent's function, allowing unpredictable behavior to reach unauthorized systems.
- Writing coarse-grained policies that span multiple systems, so a single misstep has an outsized impact.
- Skipping audit and monitoring of agent access patterns, missing the signals that indicate permission misuse.

Benefits of establishing this best practice:

- The scope of impact stays contained when an agent makes an unexpected decision.
- Clear operational boundaries make agent behavior more predictable.
- Baseline access patterns make anomalies visible instead of lost in noise.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Defense-in-depth is the frame for agent authorization. No single control is enough. AgentCore Identity, AgentCore Policy, and IAM all have roles to play, and the combination helps prevent a gap in one layer from becoming an unchecked privilege in another. Use [Amazon Bedrock AgentCore](#)

[Identity](#) to manage authentication for agent access to third-party services through OAuth and API key credentials. Use [Amazon Bedrock AgentCore Policy](#) to enforce runtime access boundaries through Cedar policies at the AgentCore Gateway boundary, independent of how the agent's LLM reasons.

For agents interacting with Amazon Bedrock models and Knowledge Bases, use IAM identity-based policies with condition keys to restrict which models each agent can invoke. Scope Memory access to designated namespaces using IAM policy conditions. Attach identity-based policies with Condition blocks that constrain access by namespace identifier and session context (e.g., `bedrock:AgentId`, `bedrock:SessionId`). With these conditions in place, agents operate within their designated memory boundaries without cross-namespace leakage. As AgentCore Memory's authorization model evolves, adopt resource-based policies when available to further simplify namespace-level grants. Avoid wildcard resources in agent IAM policies. The temptation to use `*` for convenience is the single most common reason least-privilege quietly degrades into broad access over time.

[AWS IAM Access Analyzer](#) generates least-privilege recommendations based on actual access patterns captured in CloudTrail, so policies can be tightened based on what the agent actually uses rather than what it was originally granted. CloudTrail captures the audit trail, and [Amazon Bedrock AgentCore Observability](#) detects deviations from the expected operational profile. When suspicious access patterns appear, automated responses such as permission revocation or agent quarantine help prevent the deviation from becoming an incident.

Implementation steps

1. **Configure AgentCore Identity and AgentCore Policy:** Use [Amazon Bedrock AgentCore Identity](#) for authentication and [Amazon Bedrock AgentCore Policy](#) with Cedar to enforce runtime access boundaries.
2. **Create dedicated IAM execution roles per agent:** Scope each role to specific resource ARNs and avoid wildcards.
3. **Restrict Amazon Bedrock model and Knowledge Base access with IAM condition keys:** Allow each agent only the models and knowledge bases its function requires.
4. **Audit policies with IAM Access Analyzer:** Use [AWS IAM Access Analyzer](#) to generate least-privilege recommendations from CloudTrail data and remediate overly permissive policies.
5. **Monitor access patterns and automate response:** Watch access patterns through [Amazon Bedrock AgentCore Observability](#) and configure automated permission revocation or quarantine when anomalies appear.

Resources

Related best practices:

- [AGENTRELO2-BP01 Design agents for specific and atomic tasks](#)
- [AGENTRELO2-BP03 Implement behavioral anomaly detection and monitoring](#)
- [AGENTSEC03-BP01 Implement strong authentication for agent identities](#)

Related documents:

- [Amazon Bedrock AgentCore Identity](#)
- [Amazon Bedrock AgentCore Policy](#)
- [Secure AI agents with Policy in Amazon Bedrock AgentCore](#)
- [AWS IAM Access Analyzer](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Identity and Access Management](#)

AGENTRELO2-BP03 Implement behavioral anomaly detection and monitoring

Generic logs miss what matters most for agents: decision points, tool invocations, and LLM interactions. Structured telemetry with behavioral baselines exposes anomalies early and gives operators the audit trail they need to reconstruct why an agent did what it did.

Desired outcome:

- You capture decision points, tool invocations, and LLM interactions for every agent invocation.
- You have behavioral baselines per agent and automated alarms that fire when metrics drift outside expected ranges.
- You detect behavioral drift through periodic evaluation, not only through infrastructure errors.

Common anti-patterns:

- Running generic logging that doesn't capture agent-specific decision points, leaving teams unable to understand why an agent produced an unexpected outcome.
- Operating without behavioral baselines, so there is no basis for deciding when agent behavior has actually deviated.
- Relying only on manual log review, which delays detection of reliability issues until users complain.

Benefits of establishing this best practice:

- Automated anomaly detection catches reliability issues before they cascade.
- Full execution transparency through decision-point logging speeds up root-cause analysis.
- Structured audit trails reconstruct agent decision-making for compliance and debugging.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Behavioral monitoring starts with capturing the execution path, not the final response. [Amazon Bedrock AgentCore Observability](#) provides OpenTelemetry-compatible telemetry that covers the full path of each agent invocation, from initial request through LLM inference, tool calls, memory access, and response generation. Tag traces with agent-specific metadata such as agent ID, task type, model used, and tool calls made, so filtering and analysis target the agent or failure scenario of interest.

Raw telemetry is necessary but not sufficient. Enable [Amazon Bedrock model invocation logging](#) to capture every LLM request and response, including prompts, model parameters, token counts, and latency. Without that depth, reconstructing "why did the agent choose this tool" reduces to guessing from summary metrics.

Collect agent-specific metrics over a representative period, including tool invocation frequency, output token count distribution, task completion rate, and error rate by type. Apply [Amazon CloudWatch Anomaly Detection](#) so the system learns the expected range rather than relying on fixed thresholds. Configure alarms on anomaly detection bands. Use [Amazon Bedrock AgentCore Evaluations](#) to periodically assess agent behavior against quality benchmarks so behavioral drift that doesn't show up as infrastructure errors still gets caught.

Implementation steps

1. **Enable AgentCore Observability across every invocation:** Turn on [Amazon Bedrock AgentCore Observability](#) with OpenTelemetry tracing and tag traces with agent-specific metadata.
2. **Capture full LLM request/response data:** Enable [Amazon Bedrock model invocation logging](#) for anomaly analysis and audit.
3. **Establish behavioral baselines:** Collect representative agent-specific metrics and apply [Amazon CloudWatch Anomaly Detection](#) so thresholds are learned rather than hand-tuned.
4. **Configure alarms on anomaly detection bands:** Trigger investigation workflows when metrics drift outside expected ranges.
5. **Run AgentCore Evaluations on a periodic cadence:** Use [Amazon Bedrock AgentCore Evaluations](#) to detect behavioral drift against quality benchmarks, not only infrastructure signals.

Resources

Related best practices:

- [AGENTREL02-BP01 Design agents for specific and atomic tasks](#)
- [AGENTREL02-BP02 Limit agent permissions to minimum required access](#)
- [AGENTREL08-BP02 Implement agent tracing for telemetry throughout agent processing](#)
- [AGENCOST07-BP02 Establish proactive anomaly detection for agent cost patterns](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Build trustworthy AI agents with Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock model invocation logging](#)
- [Amazon CloudWatch Anomaly Detection](#)

Related videos:

- [AWS 2025 - AgentCore Observability: Monitor and Debug with OpenTelemetry](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTREL02-BP04 Develop clear instruction protocols for agents

Ad-hoc prompts interpreted slightly differently by each model call produce unpredictable behavior, and the problem multiplies in multi-agent workflows. Standardized instruction templates, versioned prompts, and explicit handoff schemas reduce ambiguity and make regressions traceable to a specific version.

Desired outcome:

- You have a canonical system prompt template that every agent follows, covering role, capabilities, constraints, output format, and escalation behavior.
- You version prompt templates centrally and log the version used on every invocation.
- You have explicit handoff schemas for multi-agent delegation so receiving agents get unambiguous instructions.

Common anti-patterns:

- Running ad-hoc prompting without standardized formats, producing inconsistent interpretation of objectives across agents.
- Omitting explicit handoff procedures for multi-agent orchestration, leaving downstream agents to guess their role.
- Skipping prompt versioning, so rolling back a problematic change requires archaeology rather than a configuration flip.

Benefits of establishing this best practice:

- Predictable behavior through standardized instruction formats that reduce ambiguity.
- Reliable multi-agent orchestration through explicit handoff procedures and context preservation.
- Faster debugging and refinement through consistent patterns you can test systematically.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Define one structure that every system prompt follows. This structure should cover role definition, capability description, constraint specification, output format requirements, and escalation behavior. Make the template the starting point for any new agent. When every agent inherits the same structure, reviewers can check the important parts at a glance and regressions are more visible because the diffs are small.

Template storage is where versioning happens. Store prompts in a versioned configuration store so changes don't require redeployment. Assign version identifiers to every template and log the version used in every invocation through [Amazon Bedrock model invocation logging](#). When a regression appears, the version ID on the failing trace tells you exactly which template is to blame.

Handoffs need their own schema. For multi-agent orchestration, an explicit handoff message should carry the task identifier, task type, message body, execution context, deadline, and callback mechanism. Use [Amazon Bedrock AgentCore Gateway](#) to manage discovery and invocation with well-defined interface contracts. Validate new prompt versions offline using [Amazon Bedrock AgentCore Evaluations](#) to compare agent behavior before migration, and run contract tests in CI/CD including adversarial cases designed to expose prompt injection vulnerabilities.

Implementation steps

1. **Define a canonical system prompt template:** Establish a common structure for role, capabilities, constraints, output format, and escalation behavior that every agent inherits.
2. **Store prompt templates in a versioned configuration store:** Centralize management so prompt updates don't require redeployment.
3. **Design explicit handoff message schemas:** Define a canonical handoff message format for multi-agent delegation with task identifiers, message bodies, and callback mechanisms.
4. **Use AgentCore Evaluations to compare prompt versions:** Run [Amazon Bedrock AgentCore Evaluations](#) on candidate versions before migrating production traffic.
5. **Run automated contract tests in CI/CD:** Include adversarial prompt injection detection so protocol regressions don't ship.

Resources

Related best practices:

- [AGENTREL02-BP01 Design agents for specific and atomic tasks](#)

- [AGENTRELO2-BP03 Implement behavioral anomaly detection and monitoring](#)
- [AGENTRELO2-BP05 Establish tiered human oversight and approval workflows](#)

Related documents:

- [Amazon Bedrock AgentCore Evaluations](#)
- [Build reliable AI agents with Amazon Bedrock AgentCore Evaluations](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock model invocation logging](#)

Related services:

- [Amazon Bedrock AgentCore](#)

AGENTRELO2-BP05 Establish tiered human oversight and approval workflows

Uniform oversight either slows every routine action to a crawl or lets a high-consequence decision slip through unchecked. Tiering review to match the risk and reversibility of each action balances throughput with appropriate governance.

Desired outcome:

- You have agent actions classified into tiers (autonomous, notify, and approve) based on impact and reversibility.
- You have a first-pass automated review layer that filters policy-violating actions before human reviewers see them.
- You log every oversight decision with reviewer identity, rationale, and timestamp for compliance and governance reporting.

Common anti-patterns:

- Applying uniform oversight regardless of risk, creating bottlenecks for routine tasks or letting high-consequence actions slip through unchecked.

- Skipping clear escalation criteria, so some high-risk actions proceed autonomously while some low-risk actions queue for review.
- Running approval workflows without timeouts or fallback, causing agents to stall indefinitely when reviewers are unavailable.

Benefits of establishing this best practice:

- Appropriate governance for high-consequence actions without bottlenecks on routine work.
- Reduced risk from LLM stochasticity because irreversible or high-stakes decisions get human review.
- An audit trail for compliance through structured logging of oversight decisions.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Risk classification is the first design choice. Categorize agent actions into three tiers. Autonomous actions are low-risk and reversible. Notify actions are medium-risk and proceed with operator awareness. Approve actions are high-risk or irreversible and require explicit human approval. Encode the classification as Cedar policies through [Amazon Bedrock AgentCore Policy](#), so tier enforcement happens at the gateway boundary before the agent can execute. Policy-based enforcement applies the classification at runtime rather than relying on reference documentation alone.

Automated first-pass review reduces the load on human reviewers. [Amazon Bedrock Guardrails](#) intercepts agent outputs before they reach reviewers, filtering content that violates predefined policies. What reaches the human queue should be the genuinely ambiguous cases, with policy violations filtered automatically.

Approval workflows need structure, not just a pause. A structured review request should include the action description, the agent's reasoning, an impact assessment, and the execution history so the reviewer can decide quickly. Configure timeouts that escalate to secondary reviewers or fall back to safe defaults when primary reviewers are unavailable so the system handles reviewer unavailability without blocking indefinitely. Log every decision with reviewer identity, rationale, and timestamp, and monitor approval queue depth through Amazon CloudWatch to detect when reviews are accumulating. Development tools like [Kiro](#) implement this progressive autonomy pattern directly. Supervised mode reviews each action before it is applied, while autopilot mode

grants full autonomy for trusted workflows. The two modes mirror the tiered oversight model at the development layer.

Implementation steps

1. **Define a risk classification framework:** Categorize agent actions into autonomous, notify, and approve tiers based on impact and reversibility, and encode the classification as Cedar policies through [Amazon Bedrock AgentCore Policy](#).
2. **Configure Amazon Bedrock Guardrails as the automated first-pass layer:** Use [Amazon Bedrock Guardrails](#) to filter policy-violating actions before human escalation.
3. **Build structured approval workflows:** Pause execution and route review requests to reviewers. Each request should include the action description, agent reasoning, impact assessment, and execution history.
4. **Configure timeouts and escalation paths:** Handle reviewer unavailability without blocking indefinitely, with escalation to secondary reviewers or safe default fallbacks.
5. **Log every oversight decision:** Capture reviewer identity, rationale, and timestamp so the audit trail supports compliance and governance reporting.

Resources

Related best practices:

- [AGENTRELO2-BP03 Implement behavioral anomaly detection and monitoring](#)
- [AGENTRELO2-BP04 Develop clear instruction protocols for agents](#)
- [AGENTSEC04-BP02 Human-in-the-loop for critical decisions](#)
- [AGENTSUS03-BP01 Maintain organizational skills and competencies](#)

Related documents:

- [Amazon Bedrock Guardrails](#)
- [Human-in-the-loop \(HITL\) - Amazon Nova Act](#)
- [Amazon Bedrock AgentCore Policy](#)
- [Evaluating AI agents: Real-world lessons from building agentic systems at Amazon](#)

Related tools:

- [Kiro Autonomous Agent](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Agent memory and state management

Agents that classify, persist, and recover memory reliably deliver consistent behavior across sessions and maintain functionality even during component failures. Agent memory is beholden to the same reliability considerations of any data store. How do you support returning the right information, and how does the agent behave when state is unavailable?

AGENTREL03: How do you support agent memory and state remaining reliably accessible throughout the agent lifecycle?

Capability intent

- Agent memory is explicitly classified by scope and persistence, so short-term session context never contaminates long-term knowledge and retrieval is predictable across task types.
- Memory stores are redundant and have automated failover paths, so individual component failures produce degraded but usable memory rather than complete outages.
- Long-running workflows persist checkpoints at stage boundaries and resume from the last known-good state after interruption, rather than restarting from zero.
- Agents detect degraded memory conditions, transition into well-defined reduced modes, and communicate their current capabilities transparently to users and orchestration systems.
- Memory and state health are instrumented as first-class telemetry, so retrieval quality, failover activations, and checkpoint lifecycle are observable and operationally tunable.

Maturity levels

These levels summarize what each stage of maturity looks like for agent memory and state management as a whole.

Level	Name	What it looks like
1	Initial	Agent memory sits in a single undifferentiated store with no classification, no retention policy, and no failover plan. Long-running workflows restart from zero on any failure because no state is persisted between steps. Memory outages are indistinguishable from agent outages because there is no degraded mode to fall back to, and memory health is reviewed only after an incident.
2	Emerging	Teams have separated short-term and long-term memory, typically using Amazon Bedrock AgentCore Memory session-scoped and persistent namespaces. Basic retention policies are in place for short-term memory, and some long-running workflows persist checkpoint state explicitly. Redundancy and failover rely on managed service defaults, and failure modes are tested only as part

Level	Name	What it looks like
		of major release readiness reviews.
3	Defined	A documented memory taxonomy (session context, persistent knowledge, episodic records) drives classification at ingestion time, and Amazon Bedrock Knowledge Bases supplements agent memory with organizational corpora for retrieval-augmented generation. Multi-step workflows are orchestrated through AWS Step Functions or equivalent state machines with idempotent steps, so recovery is a cheap replay from the last checkpoint. Amazon Bedrock AgentCore Observability reports retrieval quality and checkpoint health, and mode transitions for graceful degradation follow written runbooks.

Level	Name	What it looks like
4	Proactive	Fail-fast logic routes memory access to fallback stores or in-process caches when primary stores misbehave , and read-through caches serve long-term memories during temporary unavailability. Mode transitions for graceful degradation (full, session-only, stateless) are automated from memory health metrics, and recovery detection returns agents to full mode without human intervention. AWS Fault Injection Service experiments against the memory layer run on a schedule, and checkpoint lifecycle is managed through TTL-based expiration rather than manual cleanup.

Level	Name	What it looks like
5	Optimized	Memory classification, retention, and checkpoint policies are tuned continuously from observability data rather than through periodic reviews. Cross-agent shared memory, tiered retrieval, and reranking strategies are recalibrated on measured retrieval quality, and chaos-engineering results drive design changes in memory topology rather than one-time fixes. Memory-related failure modes surface in telemetry and dashboards before they affect users, and the organization contributes memory and state patterns back to its internal communities of practice.

Common issues to watch for

- Teams store all agent memory in a single undifferentiated tier, so retrieval returns stale intermediate reasoning alongside stable domain facts and degrades task quality silently.
- Long-running workflows lack checkpointing, so any transient failure forces a full restart and the workflow pays its full cost every time something goes wrong.
- Workflow steps are not idempotent, so checkpoint-based replay produces duplicate side effects rather than safely resuming work.
- Memory outages are treated as agent outages because there is no degradation hierarchy, and users see confusing errors instead of clearly communicated reduced capability.
- Failover paths exist only on paper, and gaps in redundancy and recovery are discovered during production incidents because fault-injection testing isn't part of the release process.

Best practices

- [AGENTREL03-BP01 Design an information classification model to identify short-term and long-term memories](#)
- [AGENTREL03-BP02 Architect fault-tolerant memory stores with redundancy and failover](#)
- [AGENTREL03-BP03 Implement comprehensive state management and checkpoint-based recovery](#)
- [AGENTREL03-BP04 Implement graceful degradation for memory and state operations](#)

AGENTREL03-BP01 Design an information classification model to identify short-term and long-term memories

A single undifferentiated memory store blurs conversation context with durable knowledge and pulls stale or irrelevant data into active tasks. Classifying memory at ingestion time and routing each item to the right tier keeps retrieval predictable and storage costs aligned with how long each type actually needs to live.

Desired outcome:

- You have an explicit memory taxonomy distinguishing short-term session context from long-term persistent knowledge.
- Your agents classify information at ingestion time and route it to the appropriate tier with metadata tags.
- You retain each memory type according to a policy matched to its persistence requirement.

Common anti-patterns:

- Storing all memory in a single undifferentiated store, so agents retrieve stale or irrelevant items during active tasks.
- Running without retention or eviction policies for short-term memory, letting session context accumulate indefinitely.
- Skipping classification at ingestion, making targeted retrieval and appropriate retention impossible.

Benefits of establishing this best practice:

- Predictable retrieval through explicit classification that routes information to the correct store.
- Storage costs aligned with persistence requirements instead of a one-size-fits-all retention window.
- Reduced context pollution because transient session data can't contaminate long-term knowledge.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

A memory taxonomy is the foundation. At minimum, distinguish short-term (scoped to a single conversation) from long-term (durable across sessions). For complex agents, extend the taxonomy to include episodic memory (records of specific past interactions) and semantic memory (general domain knowledge). The taxonomy should be documented in accessible reference materials in addition to code, because classification decisions happen at ingestion and must be consistent across agents that write to shared memory.

[Amazon Bedrock AgentCore Memory](#) provides session-scoped and persistent memory namespaces, handling the underlying infrastructure. Use session memory for short-term context scoped to a single conversation, persistent memory for cross-session knowledge, and shared memory for facts multiple agents consume. [Amazon Bedrock Knowledge Bases](#) handles RAG over external document corpora, policy documents, product catalogs, and domain reference material that supplement agent memory with organizational knowledge.

Classification logic belongs in a dedicated component of the agent's processing flow, not scattered across prompts. Evaluate each piece of information against source (conversation turn vs. task outcome), temporal scope (session-specific vs. cross-session), and content type (procedural vs. factual). Route accordingly and tag with metadata so later retrieval can filter precisely. Monitor memory access patterns through [Amazon Bedrock AgentCore Observability](#) to catch misclassified memories. The signal is usually unexpectedly high retrieval rates from the wrong tier.

Implementation steps

1. **Define a memory taxonomy:** Document classification criteria for each memory type (session context, persistent knowledge, episodic records) and the retention policy that fits each.
2. **Configure AgentCore Memory namespaces:** Provision [Amazon Bedrock AgentCore Memory](#) with session-scoped and persistent namespaces that map to the taxonomy.

3. **Implement classification logic at ingestion:** Evaluate each item against the taxonomy criteria and route to the appropriate tier with metadata tags.
4. **Use Amazon Bedrock Knowledge Bases for external corpora:** Supplement agent memory with [Amazon Bedrock Knowledge Bases](#) for RAG over organizational knowledge.
5. **Monitor access patterns to detect classification errors:** Use [Amazon Bedrock AgentCore Observability](#) to spot unexpectedly high retrieval rates from wrong tiers.

Resources

Related best practices:

- [AGENTRELO3-BP02 Architect fault-tolerant memory stores with redundancy and failover](#)
- [AGENTRELO3-BP03 Implement comprehensive state management and checkpoint-based recovery](#)
- [AGENTRELO3-BP04 Implement graceful degradation for memory and state operations](#)
- [AGENTSUS02-BP01 Optimize context management and memory utilization](#)

Related documents:

- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Memory: Building context-aware agents](#)
- [Building smarter AI agents: AgentCore long-term memory deep dive](#)
- [Amazon Bedrock Knowledge Bases](#)
- [Strands Agents Session Management](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Memory](#)
- [AWS re:Invent 2024 - Make agents remember with AgentCore Memory \(AIM331\)](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore - Lab 2: Memory](#)
- [Diving Deep into Bedrock AgentCore - Memory](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)

AGENTREL03-BP02 Architect fault-tolerant memory stores with redundancy and failover

Memory failures don't need to mean agent failures. With redundancy, fallback paths, and a discipline of testing failover under controlled conditions, an agent keeps serving reduced-capability responses until its primary stores recover instead of becoming completely unavailable.

Desired outcome:

- You have primary memory infrastructure with built-in durability and availability, backed by explicit fallback stores for degraded operation.
- You have fail-fast logic on memory access that routes to fallback when primary stores are unavailable.
- You exercise failover regularly in non-production environments to validate degraded-mode behavior.

Common anti-patterns:

- Running memory stores as single points of failure without replication or failover, causing complete memory loss during outages.
- Leaving failover manual, so recovery waits on operators and extends agent downtime.
- Skipping failover testing, discovering the gaps only when production incidents force the issue.

Benefits of establishing this best practice:

- Downtime drops because automated failover takes over before operators can intervene.
- Agents keep behaving consistently during memory store failures through graceful degradation.
- Memory replication across Availability Zones helps protect against data loss.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock AgentCore Memory](#) provides managed memory infrastructure with built-in durability and availability, so the default path is already fault-tolerant. The design work is on the degraded path: what does the agent do when even the managed store is briefly unreachable, or when a custom store sits alongside it? Fail-fast logic on memory access is the first answer. When a store shows elevated error rates, the caller stops waiting and routes to a fallback. For short-term memory, that fallback is an in-process cache. For long-term memory, it is a read-through cache of frequently accessed items.

For agents running on [Amazon Bedrock AgentCore Runtime](#), the runtime's managed session storage persists filesystem-level state across stop and resume cycles. For workflow-stage-aware checkpointing with redrive from specific failure points, use [AWS Step Functions](#) or framework-level orchestration such as LangGraph with AgentCore Memory. The choice depends on how granular the recovery needs to be. Step Functions gives you durability for each step, while managed session storage gives you whole-agent durability at session boundaries.

Regular testing validates that failover mechanisms work as designed. [AWS Fault Injection Service](#) simulates memory store failures in non-production environments so you can validate that failover mechanisms activate correctly and agents continue operating in degraded mode. Document expected behavior for each failure scenario and compare observed behavior against the expectations every time you run the test. Drift between what you expect and what actually happens is the signal that a regression slipped in.

Implementation steps

1. **Use AgentCore Memory as the primary managed store:** Default to [Amazon Bedrock AgentCore Memory](#) for its built-in durability and availability.
2. **Implement fail-fast logic for memory access:** Detect elevated error rates on memory calls and route to fallback stores.
3. **Maintain in-process fallback caches for short-term memory:** Keep current sessions moving through a last-resort cache that lets the task complete.
4. **Implement read-through caching for long-term memory:** Serve cached copies of frequently accessed items during temporary unavailability.
5. **Test failover with AWS Fault Injection Service:** Use [AWS Fault Injection Service](#) to validate degraded-mode behavior against documented expectations on a regular schedule.

Resources

Related best practices:

- [AGENTREL01-BP02 Establish modular, fault-isolated layers](#)
- [AGENTREL03-BP01 Design an information classification model to identify short-term and long-term memories](#)
- [AGENTREL03-BP03 Implement comprehensive state management and checkpoint-based recovery](#)
- [AGENTREL03-BP04 Implement graceful degradation for memory and state operations](#)

Related documents:

- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Runtime](#)
- [AWS Fault Injection Service](#)
- [AWS fail-fast pattern](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Memory tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Fault Injection Service](#)

AGENTREL03-BP03 Implement comprehensive state management and checkpoint-based recovery

Long-running workflows without checkpoints pay the full restart cost for every failure, no matter how late it happens. Persisting state at natural boundaries and designing every step to be idempotent lets an agent resume from the last completed checkpoint rather than redo work.

Desired outcome:

- You have workflow state persisted at regular checkpoints, so interruptions resume from the last completed point rather than the beginning.
- You have idempotent workflow steps that produce the same result when replayed with the same input.
- You have a checkpoint lifecycle with TTL-based expiration and explicit cleanup after completion.

Common anti-patterns:

- Running long-duration agent workflows without intermediate state persistence, forcing complete restarts on any failure.
- Implementing checkpoints without idempotency guarantees, producing data corruption or duplicate side effects on resume.
- Skipping checkpoint cleanup, accumulating storage indefinitely.

Benefits of establishing this best practice:

- Workflow restart cost drops because resume starts from the last checkpoint.
- Duplicate work is prevented through idempotent checkpoint-based recovery.
- Compute efficiency improves because recovery avoids redundant recomputation.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Checkpointing is only useful if recovery is safe, and recovery is only safe if steps are idempotent. An idempotent step produces the same result whether it runs once or five times with the same input, which means a retry or a resume doesn't add duplicate side effects. This constraint shapes everything downstream. External calls need idempotency keys, state mutations need conditional writes, and event emissions need deduplication logic. Without idempotency guarantees, checkpoint-based recovery can produce duplicate side effects or data corruption. Design each step to be idempotent before implementing checkpointing.

Runtime choice determines how much checkpointing discipline you need to build yourself. [Amazon Bedrock AgentCore Runtime](#) supports long-running workloads with managed session storage that persists filesystem state across stop and resume cycles, which covers the coarse-grained case. For workflow-stage-aware checkpointing with redrive from specific failure points, orchestrate through

[AWS Step Functions](#). Step Functions persists execution state at every transition and enables restart from the point of failure rather than from the beginning. For dynamic workflows driven by supervisor agents, callback patterns pause execution while the supervisor decides the next action, preserving state persistence benefits.

Lifecycle management keeps the checkpoint store from growing without bound. TTL-based expiration handles the common case: workflows that never complete eventually age out. Explicit cleanup after successful completion reclaims space immediately. Use [Amazon Bedrock AgentCore Memory](#) to persist checkpoint state and specification context for agents requiring custom checkpointing. Monitor checkpoint store health through [Amazon Bedrock AgentCore Observability](#) so storage growth or access latency surfaces before recovery starts failing.

Implementation steps

1. **Deploy agents on AgentCore Runtime with managed session storage:** Use [Amazon Bedrock AgentCore Runtime](#) for filesystem-level state persistence across stop and resume cycles.
2. **Orchestrate multi-step workflows through Step Functions:** Use [AWS Step Functions](#) for state persistence at every transition with redrive capability from the point of failure.
3. **Design every workflow step to be idempotent:** Require idempotency keys on external calls and conditional writes on state mutations so retries and resumes don't introduce duplicate side effects.
4. **Use AgentCore Memory for custom checkpoint state:** Persist checkpoint state and specification context through [Amazon Bedrock AgentCore Memory](#) for agents with bespoke checkpointing needs.
5. **Implement checkpoint lifecycle management:** Set TTL-based expiration and explicit cleanup after successful completion so the checkpoint store stays bounded.

Resources

Related best practices:

- [AGENTREL03-BP01 Design an information classification model to identify short-term and long-term memories](#)
- [AGENTREL03-BP02 Architect fault-tolerant memory stores with redundancy and failover](#)
- [AGENTREL07-BP01 Design workflows in stages with incremental recovery](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [AWS Step Functions](#)
- [Amazon Bedrock AgentCore Memory](#)
- [Build resilient generative AI agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Step Functions](#)

AGENTREL03-BP04 Implement graceful degradation for memory and state operations

Treating every memory failure as fatal turns recoverable issues into total outages. An explicit degradation hierarchy with documented modes and automatic recovery keeps agents partially useful and transparent about their reduced state.

Desired outcome:

- You have a memory degradation hierarchy with distinct operational modes and documented behaviors for each.
- You transition modes automatically based on memory health signals, and recover to full mode when stores return.
- You communicate degradation state to users and orchestration systems so they know what to expect.

Common anti-patterns:

- Treating all memory failures as fatal, producing complete unavailability for conditions that could be handled with reduced functionality.
- Failing to communicate degraded memory state to users, who then get confused when responses lack expected context.
- Implementing degradation without a recovery path, leaving agents in degraded mode indefinitely after primary stores return.

Benefits of establishing this best practice:

- Partial service availability persists through memory store failures instead of becoming a full outage.
- Users and orchestrators see transparent indications of current capability.
- Full capability returns automatically when memory stores come back online.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Start with the mode hierarchy. Three modes cover most agent workloads. Full mode has all memory tiers available. Session-only mode operates without long-term memory, using only session context. Stateless mode has both tiers unavailable and processes each request independently. For each mode, define the agent's behavior explicitly:

- In session-only mode, inform users that previous session context is unavailable
- In stateless mode, request all necessary context within the current interaction.

Without this definition, the fallback path is whatever the code happens to do, which is rarely what you want during an incident.

Health signals drive the transitions. [Amazon Bedrock AgentCore Observability](#) and Amazon CloudWatch together track memory store availability and error rates. Configure automated mode transitions when health degrades below thresholds you set in advance, and automatic recovery when stores return. For short-term memory degradation, in-process fallback caches let the current session continue. [Amazon Bedrock AgentCore Memory](#) session management maintains working context even when long-term stores are degraded, which keeps most conversations coherent through a partial outage.

Communicate degradation state through structured status indicators so users and downstream systems understand the current limitations. "I don't have access to your previous conversations right now" is a better experience than an agent that silently pretends it does and hallucinates. The same signals feed orchestration systems that might choose to route requests elsewhere or surface a banner to users.

Implementation steps

1. **Define a memory degradation hierarchy with documented modes:** Specify full, session-only, and stateless modes, with the agent behavior each mode dictates.
2. **Implement automated mode transitions:** Trigger transitions through health metrics from [Amazon Bedrock AgentCore Observability](#) and Amazon CloudWatch.
3. **Maintain in-process fallback caches for short-term memory:** Allow active sessions to continue when short-term memory degrades.
4. **Communicate degradation state to users:** Surface structured status indicators so users see the reduced capability instead of guessing.
5. **Configure automatic recovery detection:** Return agents to full mode when stores become available, without operator intervention.

Resources

Related best practices:

- [AGENTRELO3-BP01 Design an information classification model to identify short-term and long-term memories](#)
- [AGENTRELO3-BP02 Architect fault-tolerant memory stores with redundancy and failover](#)
- [AGENTRELO8-BP04 Track agent memory utilization metrics](#)

Related documents:

- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Build resilient generative AI agents](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Memory tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)

- [Amazon CloudWatch](#)

Multi-agent orchestration

Multi-agent systems that implement centralized coordination, capability-based routing, and pre-defined fallback chains execute tasks reliably even when individual agents fail. Multi-agent workflows must be highly orchestrated and controlled to help prevent unreliable agent executions from disrupting an entire workflow.

AGENTREL04: How do you orchestrate multi-agent systems to reliably execute tasks?

Capability intent

- Conflict resolution is concentrated in a dedicated arbiter that acts only when coordination is needed, so specialized agents operate independently without negotiating every disagreement peer-to-peer.
- Agents are described in a structured capability taxonomy that drives deterministic routing and automatic substitution when a preferred agent is unavailable.
- Each critical agent in a collaborative workflow has an explicit, ordered fallback chain with documented quality trade-offs, so individual failures produce reduced capability rather than workflow collapse.
- The control plane itself is redundant, durable, and loosely coupled to agents, so coordination infrastructure is at least as reliable as the agents it coordinates.
- Arbitration decisions, routing outcomes, fallback activations, and control-plane health are all observable as first-class telemetry, and failure modes are validated through regular fault-injection and disaster recovery exercises.

Maturity levels

These levels summarize what each stage of maturity looks like for multi-agent orchestration as a whole.

Level	Name	What it looks like
1	Initial	Agents coordinate directly with each other, so conflicts produce deadlocks, circular dependencies, or inconsistent state. Orchestration hard-codes specific agent identifiers, fallback paths are absent, and the control plane often runs as a single instance with in-memory state. Multi-agent failures are diagnosed only after incidents.
2	Emerging	A central arbiter exists for critical conflict resolution, and agents are registered in a simple catalog that orchestrators consult for routing. Basic fallback logic handles primary agent failures, although fallback behavior is one-time per workflow. The control plane uses managed services such as Amazon Bedrock AgentCore Runtime for execution, but workflow state isn't persisted end to end.
3	Defined	Arbitration policies are externalized from the arbiter binary and stored in Parameter Store , a capability of AWS Systems Manager or Amazon DynamoDB , with human escalation

Level	Name	What it looks like
		through Amazon SNS for unresolvable conflicts. Agents are registered in Amazon Bedrock AgentCore Registry with structure d capability metadata and discovered through semantic search. Fallback chains are documented per critical agent, and workflow orchestration uses AWS Step Functions for durable state.
4	Proactive	The arbiter is event-driven through Amazon EventBridge, activating only for conflict resolution rather than mediating every message. Capability registration is automated in CI/CD so the registry stays aligned with deployed state. Proactive health checking through the Amazon Bedrock AgentCore Runtime /ping endpoint drives fallback activation without waiting for timeouts, and AWS Fault Injection Service exercises validate fallback chains on a schedule. Agents tolerate brief control-plane outages because they are designed to complete in-flight work independently.

Level	Name	What it looks like
5	Optimized	Arbitration policies, routing decisions, and fallback tiers are recalibrated continuously from Amazon Bedrock AgentCore Observability telemetry rather than through periodic review cycles. Contention hotspots and capability gaps surface in Amazon CloudWatch Contributor Insights and drive targeted redesign of coordination protocols. Disaster recovery exercises are routine, control-plane failover is provably automated, and the organization contributes multi-agent orchestration patterns back to its internal communities of practice.

Common issues to watch for

- Teams let agents coordinate peer-to-peer without a dedicated arbiter, producing deadlocks and inconsistent outcomes whenever agents contend for the same resource.
- Orchestration hard-codes specific agent identifiers, so routing can't adapt when agents are replaced or become unavailable, and agent changes require orchestration code changes.
- Fallback chains exist for critical agents but are never exercised, so gaps in coverage are discovered during production incidents rather than fault-injection tests.
- The control plane is treated as a single point of failure with in-memory state, so its failure loses coordination context and disrupts every agent at once.
- Aggregate coordination cost and quality are the only metrics tracked, so contention hotspots and capability-matching failures stay invisible until they dominate the user-visible experience.

Best practices

- [AGENTRELO4-BP01 Implement the arbiter agent pattern for coordinated multi-agent systems](#)
- [AGENTRELO4-BP02 Classify agents with a thorough capability taxonomy](#)
- [AGENTRELO4-BP03 Implement fallback mechanisms and graceful degradation for collaborative workflows](#)
- [AGENTRELO4-BP04 Implement resilient control planes for agent coordination](#)

AGENTRELO4-BP01 Implement the arbiter agent pattern for coordinated multi-agent systems

Peer-to-peer coordination among agents produces deadlocks and conflicting actions at scale. A dedicated arbiter that activates only for conflict resolution preserves agent autonomy for normal work while providing a single authoritative place to resolve contention over shared resources.

Desired outcome:

- You have an arbiter agent that activates for conflict resolution while leaving routine coordination to the specialized agents involved.
- You store conflict resolution policies in a configuration store so policy updates don't require arbiter redeployment.
- You escalate unresolvable conflicts to human review automatically.

Common anti-patterns:

- Letting agents coordinate directly without a central arbiter, creating circular dependencies and deadlocks when requirements conflict.
- Embedding conflict resolution logic inside specialized agents, producing inconsistent arbitration across the system.
- Routing every agent interaction through the arbiter, turning it into a performance bottleneck.

Benefits of establishing this best practice:

- Deadlocks and conflicting actions get resolved by a single authority instead of leaking into business logic.
- Consistent workflow behavior comes from arbitration policies applied uniformly across the fleet.

- Specialized agents stay independent because coordination logic is implemented in the arbiter, not in them.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Design the arbiter as an event-driven component rather than a synchronous dispatcher. The arbiter subscribes to coordination events through Amazon EventBridge or Amazon SQS, activates only when a specialized agent explicitly requests arbitration, and publishes decisions back to the requesting agents. Synchronous arbitration creates a bottleneck that reduces agent autonomy.

Policy shape matters as much as arbiter placement. Three categories cover most real coordination conflicts. Priority-based ordering handles resource contention, confidence scoring resolves conflicting decisions, and rollback instructions address constraint violations. Store the policies in Parameter Store, a capability of AWS Systems Manager or Amazon DynamoDB so they can be updated without redeploying the arbiter. When a policy turns out to be wrong, the fix lands quickly rather than waiting for a deployment cycle. For real-time arbitration needs, implement the arbiter as a synchronous service invoked through [Amazon Bedrock AgentCore Gateway](#), accepting that those paths will be slower but reserving them for cases where asynchrony isn't acceptable.

Monitoring the arbiter keeps coordination healthy. Track conflict frequency, arbitration latency, and escalation rates through [Amazon Bedrock AgentCore Observability](#). Use Amazon CloudWatch Contributor Insights to identify which agent pairs or resource types are most frequently involved in conflicts. Those are the pairs where a coordination protocol redesign pays off the most. Unresolvable conflicts escalate to human review through Amazon SNS notifications or a ticketing integration so they are not silently dropped.

Implementation steps

1. **Design the arbiter as an event-driven agent on AgentCore Runtime:** Deploy the arbiter on [Amazon Bedrock AgentCore Runtime](#) and have it subscribe to coordination events through EventBridge, activating only for conflict resolution.
2. **Create conflict resolution policies in a configuration store:** Store rules for resource contention, conflicting decisions, and constraint violations in Parameter Store (a capability of AWS Systems Manager) or Amazon DynamoDB.

3. **Publish decisions through EventBridge or AgentCore Gateway:** Route arbitration decisions back to the affected agents through EventBridge for asynchronous paths or direct invocation through [Amazon Bedrock AgentCore Gateway](#) for synchronous needs.
4. **Configure CloudWatch alarms and Contributor Insights:** Alarm on abnormal conflict frequency and use Contributor Insights to expose the most contention-prone agent pairs and resource types.
5. **Implement escalation to human review:** Route unresolvable conflicts to Amazon SNS notifications or a ticketing integration so they reach an operator.

Resources

Related best practices:

- [AGENTRELO4-BP02 Classify agents with a thorough capability taxonomy](#)
- [AGENTRELO4-BP03 Implement fallback mechanisms and graceful degradation for collaborative workflows](#)
- [AGENTRELO4-BP04 Implement resilient control planes for agent coordination](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway](#)
- [Agentic AI patterns and workflows on AWS](#)
- [Build resilient generative AI agents](#)
- [Customize agent workflows with advanced orchestration techniques using Strands Agents](#)
- [Multi Agent Collaboration with Strands](#)

Related videos:

- [Breaking multi-agent silos: A2A + MCP in action with Strands Agents](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Multi-agent tutorials](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [Amazon EventBridge](#)

AGENTREL04-BP02 Classify agents with a thorough capability taxonomy

Orchestrators that pick agents by hardcoded identifiers can't adapt when the preferred agent is unavailable or when a new equivalent arrives. A structured capability taxonomy gives the orchestrator a basis for routing decisions, and substitution becomes automatic rather than a redeployment.

Desired outcome:

- You have every agent registered with capability categories, skills, input/output constraints, performance profiles, and dependencies.
- Your orchestrators consult the registry to select agents rather than hardcoding identifiers.
- You keep the registry current through the CI/CD pipeline so it reflects the deployed state.

Common anti-patterns:

- Hardcoding agent selection in orchestration logic without consulting a capability registry, reducing the risk of dynamic routing.
- Defining capabilities at too coarse a granularity, missing the nuances of skills, limitations, and resource requirements.
- Letting the capability registry drift from the deployed state when agents are updated.

Benefits of establishing this best practice:

- Deterministic task routing through structured capability matching rather than trial and error.

- Automatic agent substitution when preferred agents are unavailable, without manual reconfiguration.
- Fewer task failures from capability mismatches through precise capability-to-task matching.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

A capability registry is only useful if it stays current. To keep it current, integrate registration into the deployment pipeline. An agent reaches production by going through a step that also updates its entry in [Amazon Bedrock AgentCore Registry](#). Skip that step and the registry becomes a documentation artifact that diverges from reality within weeks.

AgentCore Registry's semantic capability search makes the registry useful at runtime. Orchestrators discover agents through natural language queries that match task requirements to agent capabilities without hardcoded routing logic. The quality of search results depends heavily on the quality of the record descriptions. Descriptions that explain what each agent does and the problems it solves in plain language produce good matches. Descriptions that read like function signatures produce poor matches.

Routing builds on top of registry data. The capability matching layer accepts a task specification and returns ranked agents that satisfy the requirements, ordered by match quality and operational suitability. Use [Amazon Bedrock AgentCore Gateway](#) to route invocations to the selected agent. Monitor routing effectiveness through [Amazon Bedrock AgentCore Observability](#). Capability match failures and routing decisions that result in errors are the signals you use to find capability gaps.

Implementation steps

1. **Register every agent in AgentCore Registry:** Populate [Amazon Bedrock AgentCore Registry](#) with capability categories, skills, constraints, and performance profiles for each agent.
2. **Automate registration in the CI/CD pipeline:** Make the deployment step that updates production also update the registry so the two stay in sync.
3. **Use AgentCore Registry's hybrid search to match tasks to agents:** Write record descriptions in natural language that explain what each agent does and the problems it solves, so semantic search produces accurate matches.
4. **Configure orchestrators to consult the registry:** Replace hardcoded agent identifiers with registry lookups.

5. **Monitor routing effectiveness:** Use [Amazon Bedrock AgentCore Observability](#) to find capability mismatches and gaps.

Resources

Related best practices:

- [AGENTRELO4-BP01 Implement the arbiter agent pattern for coordinated multi-agent systems](#)
- [AGENTRELO4-BP03 Implement fallback mechanisms and graceful degradation for collaborative workflows](#)
- [AGENTRELO4-BP04 Implement resilient control planes for agent coordination](#)

Related documents:

- [Amazon Bedrock AgentCore Registry](#)
- [The future of managing agents at scale: AWS Agent Registry now in preview](#)
- [Amazon Bedrock AgentCore Gateway](#)

Related videos:

- [AWS 2025 - AgentCore Registry: Discover, Govern, and Reuse AI Agents at Scale](#)

Related services:

- [Amazon Bedrock AgentCore](#)

AGENTRELO4-BP03 Implement fallback mechanisms and graceful degradation for collaborative workflows

One unavailable agent should not take down an entire workflow. Pre-defined fallback chains let orchestrators swap in alternatives, preserving forward progress with reduced quality rather than a complete stall.

Desired outcome:

- You have fallback chains for each critical agent with ordered alternatives and documented quality trade-offs.
- You check agent health proactively and skip unavailable agents rather than waiting for timeout.
- You communicate degradation to downstream systems through structured events so their behavior can adapt.

Common anti-patterns:

- Designing multi-agent workflows without fallback paths, so one failed agent halts the entire workflow.
- Implementing fallbacks that silently degrade quality without telling users or downstream systems.
- Skipping fallback testing, discovering gaps only during production incidents.

Benefits of establishing this best practice:

- Partial workflow functionality persists when an individual agent fails.
- Transparent degradation reaches users and downstream systems so they can adapt.
- Faster workflow completion through pre-defined fallback paths that activate automatically.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Fallback chains give the orchestrator somewhere to go when the preferred agent is down. Each chain is an ordered sequence. First, a secondary agent with equivalent capabilities. Then, a simplified agent with reduced capabilities. Next, a cached result from a previous execution. Finally, a graceful failure response. The ordering matters because it captures the quality trade-off. The first few alternatives preserve most of the functionality, and the later ones accept larger degradation in exchange for keeping the workflow moving at all. Document the quality impact of each level so orchestrators pick the best available option rather than the first technically viable one.

Proactive health checking keeps fallback latency low. Without it, the orchestrator waits for the preferred agent to time out before trying the fallback, which stacks agent-level latency penalties on top of the workflow. Check [Amazon Bedrock AgentCore Observability](#) metrics and [Amazon](#)

[Bedrock AgentCore Runtime](#)'s /ping endpoint before invocation. When an agent reports degraded health, skip it and move directly to the next alternative.

When a fallback activates, publish a structured degradation event that identifies the failed agent, the activated fallback, and the capability impact. Downstream systems subscribe and adapt by flagging outputs for additional review, displaying degradation notices to users, or routing around the affected workflow entirely. Validate fallback mechanisms through chaos engineering using [AWS Fault Injection Service](#). Inject agent failures in non-production environments to confirm fallback chains activate correctly and workflows complete with expected degraded outputs.

Implementation steps

1. **Design fallback chains for each critical agent:** Define an ordered sequence of alternatives with documented quality trade-offs at each level.
2. **Implement proactive health checking before invocation:** Check [Amazon Bedrock AgentCore Observability](#) metrics and the [Amazon Bedrock AgentCore Runtime](#) /ping endpoint, and skip agents reporting degraded health.
3. **Configure fallback transitions in the orchestration layer:** Distinguish transient failures (retry first) from permanent failures (immediate fallback).
4. **Publish structured degradation events when fallbacks activate:** Emit events for downstream systems to consume so the rest of the environment can adapt.
5. **Validate fallback mechanisms through chaos engineering:** Use [AWS Fault Injection Service](#) to inject agent failures on a regular schedule and confirm the chains still work.

Resources

Related best practices:

- [AGENTRELO4-BP01 Implement the arbiter agent pattern for coordinated multi-agent systems](#)
- [AGENTRELO4-BP02 Classify agents with a thorough capability taxonomy](#)
- [AGENTRELO4-BP04 Implement resilient control planes for agent coordination](#)
- [AGENTRELO8-BP03 Architect agent systems with resource isolation and contention mitigation](#)

Related documents:

- [Amazon Bedrock AgentCore Registry](#)

- [Amazon Bedrock AgentCore Runtime](#)
- [Build resilient generative AI agents](#)
- [Multi-agent collaboration with Strands](#)
- [AWS Fault Injection Service](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Fault Injection Service](#)
- [Amazon CloudWatch](#)

AGENTREL04-BP04 Implement resilient control planes for agent coordination

A control plane that fails takes every agent with it. Applying the same reliability principles used on agents, redundancy, durable state, and loose coupling, to the coordination infrastructure keeps workflows running during brief outages and preserves state across restarts.

Desired outcome:

- You deploy agents on managed, highly available execution infrastructure with multi-AZ redundancy.
- You persist workflow state durably so the control plane can recover without losing progress.
- You design agents to complete in-flight work during brief control plane outages rather than failing immediately.

Common anti-patterns:

- Implementing the control plane as a single point of failure without redundancy, so outages take down the entire multi-agent system.
- Holding control plane state in ephemeral memory, losing coordination state whenever the control plane restarts.
- Coupling agent execution tightly to control plane availability, reducing the ability for agents to complete in-progress work during brief outages.

Benefits of establishing this best practice:

- Multi-agent workflows keep running through control plane component failures.
- Durable state persistence reduces workflow state loss during outages.
- Loose coupling keeps agents productive during brief control plane unavailability.

Level of risk exposed if this best practice is not established: High

Implementation guidance

AgentCore Runtime is designed as a regional service. Architect your agents assuming the underlying compute is distributed across Availability Zones, but validate this assumption for your specific workload by confirming endpoint behavior during AZ impairment (e.g., using AZ-isolated canary deployments). Don't rely solely on service-level redundancy. Implement your own cross-AZ resilience patterns (multi-AZ deployment of agent orchestrators, regional failover for stateful components) to maintain availability targets independent of any single service's internal architecture.

Durable state keeps recovery clean. [AWS Step Functions](#) persists execution state at every transition, providing built-in retry, error handling, and resume-from-failure semantics for workflows that need explicit state machines. Without durable state, every control plane restart requires agents to recover the coordination context themselves, which is error-prone and often incomplete.

Loose coupling is the third property, and the hardest to build in after the fact. Agents should complete in-flight tasks independently if the control plane is briefly unavailable, rather than failing immediately on loss of connectivity. Heartbeat mechanisms let agents periodically report status so the control plane can detect missed heartbeats and reassign tasks, catching the cases where an agent has genuinely stopped responding. Monitor the AgentCore Runtime /ping endpoint for each agent as the liveness signal, and configure the orchestration layer to reassign tasks when agents stop responding. Composite alarms through [Amazon Bedrock AgentCore Observability](#) aggregate signals across coordination components. Regular disaster recovery exercises validate that automated failover actually works when you need it.

Implementation steps

1. **Deploy agents on AgentCore Runtime:** Use [Amazon Bedrock AgentCore Runtime](#) as the primary execution infrastructure for its built-in multi-AZ redundancy.

2. **Use AWS Step Functions for explicit workflow state machines:** Run workflows on [AWS Step Functions](#) for durable state persistence and automatic recovery.
3. **Use AgentCore Gateway for agent discovery and invocation:** Route agent calls through [Amazon Bedrock AgentCore Gateway](#) for its built-in availability characteristics.
4. **Design agents to complete in-flight work during brief control plane outages:** Avoid patterns that require constant control plane connectivity.
5. **Implement agent liveness detection through the AgentCore Runtime /ping endpoint:** Monitor the endpoint for each agent and reassign tasks through the orchestration layer when agents stop responding.
6. **Run regular disaster recovery exercises:** Use [Amazon Bedrock AgentCore Observability](#) composite alarms and periodic DR drills to validate automated failover.

Resources

Related best practices:

- [AGENTRELO4-BP01 Implement the arbiter agent pattern for coordinated multi-agent systems](#)
- [AGENTRELO4-BP02 Classify agents with a thorough capability taxonomy](#)
- [AGENTRELO4-BP03 Implement fallback mechanisms and graceful degradation for collaborative workflows](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [AWS Step Functions](#)
- [Agentic AI patterns and workflows on AWS](#)
- [Build resilient generative AI agents](#)

Related videos:

- [AWS re:Invent 2024 - Architecting scalable and secure agentic AI with AgentCore \(AIM431\)](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Step Functions](#)
- [Amazon CloudWatch](#)

Agent cognition

Agents that ground their reasoning in current, accurate information and adapt based on structured feedback deliver reliable cognition even as task requirements and data distributions evolve.

Agentic cognition engines must be provided with appropriate information at all layers of workflow execution to reliably execute tasks.

AGENTREL05: How do you implement reliable agent cognition that accesses the right data at the right time?

Capability intent

- Agent reasoning is decomposed into modular stages with explicit interfaces, so a failure in one stage produces stage-scoped fallback rather than complete cognition failure.
- Context retrieval and model inference each use tiered strategies, so reduced but useful cognition remains available when the primary retrieval tier or primary model is degraded.
- Agent cognition is grounded in retrieved real-world information through retrieval-augmented generation and real-time tool calls, so hallucination rates are lower and outputs reflect current facts.
- Agent behavior improves through evaluation-driven cycles where feedback is collected, outcomes are assessed, and prompt or configuration changes are validated offline before deployment, rather than through runtime self-modification.
- Per-stage health, retrieval quality, and evaluation results are observable as first-class telemetry, so the weakest links in cognition surface in dashboards rather than user complaints.

Maturity levels

These levels summarize what each stage of maturity looks like for agent cognition as a whole.

Level	Name	What it looks like
1	Initial	Agent cognition is a monolithic pipeline where any component failure causes complete cognition failure. Agents rely on model training data for domain knowledge , with no retrieval grounding and no feedback collection. Systematic errors are discovered through user complaints, and adaptation happens through one-time prompt tweaks applied directly to production.
2	Emerging	Reasoning is broken into identifiable stages, and basic retrieval grounds agent output in organizational knowledge through Amazon Bedrock Knowledge Bases . Teams capture some feedback signals, and periodic evaluations compare agent outputs against a small set of golden-path examples. Fallbacks exist for obvious failure paths but are not exercised systematically.
3	Defined	Each reasoning stage has explicit input and output schemas and independent error handling, deployed on Amazon Bedrock AgentCore

Level	Name	What it looks like
		Runtime . Multi-tier feedback (action-level, task-level, session-level) is captured and stored alongside task records, and Amazon Bedrock AgentCore Evaluations assesses agent performance against representative task sets. Knowledge base synchronization pipelines are automated, and chunking and reranking strategies are tuned per content type.
4	Proactive	Automatic cutoffs between stages activate stage-specific fallbacks when error rates exceed thresholds, and tiered context retrieval plus Amazon Bedrock cross-region inference keep cognition available when primary tiers degrade. Multimodal preprocessing through Amazon Bedrock Data Automation is a distinct stage with its own fallbacks. Offline prompt optimization workflows deploy validated improvements through gradual rollout, and knowledge freshness thresholds produce observable alerts.

Level	Name	What it looks like
5	Optimized	Per-stage quality, retrieval effectiveness, and evaluation outcomes feed a continuous improvement loop that tunes chunking, reranking, model tier selection, and prompt strategies. Hallucination rates, grounding coverage, and per-stage fallback activation are tracked as key reliability indicators, and cognition architecture decisions are driven by observability data rather than intuition. The organization contributes agent-cognition patterns and measurements back to its internal communities of practice.

Common issues to watch for

- Teams build agent cognition as a monolithic pipeline, so any stage failure causes a complete cognition failure and debugging has to pick apart a single large black box after the fact.
- Agents rely only on model training data for domain-specific knowledge and produce confidently wrong answers about current facts, because retrieval grounding isn't in place.
- Retrieval is treated as a hard availability dependency, so knowledge base outages become agent outages instead of clearly communicated degraded grounding.
- Agents are deployed without multi-tier feedback collection, so systematic errors are discovered from user complaints rather than from telemetry.
- Prompt and configuration changes are applied at runtime without offline evaluation, producing unpredictable regressions when feedback is noisy or evaluations are skipped.

Best practices

- [AGENTREL05-BP01 Design modular, fault-tolerant agentic reasoning components](#)
- [AGENTREL05-BP02 Facilitate reliable adaptation through evaluation-driven improvement cycles](#)
- [AGENTREL05-BP03 Ground agent cognition in real information](#)

AGENTREL05-BP01 Design modular, fault-tolerant agentic reasoning components

A monolithic reasoning pipeline fails completely whenever any stage fails. Splitting cognition into modular stages with clear interfaces and stage-specific fallbacks lets an agent keep reasoning, with reduced quality, even when one stage is degraded.

Desired outcome:

- You have the reasoning pipeline decomposed into modular stages with explicit input/output schemas.
- You have stage-specific fallbacks that activate automatically when error rates climb.
- You log the retrieval tier and model tier used in each invocation so quality analysis is possible after the fact.

Common anti-patterns:

- Running agent cognition as a monolithic pipeline where any component failure causes complete cognition failure.
- Skipping interfaces between reasoning components, reducing the ability for independent testing and replacement.
- Treating all reasoning components as equally critical without distinguishing essential from quality-enhancing components.

Benefits of establishing this best practice:

- Partial cognition survives individual component failures through modular fault isolation.
- Reasoning components can be optimized or replaced independently, without full pipeline rewrites.
- Clear component boundaries isolate the source of errors and speed up debugging.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The first architectural decision is where the stage boundaries go. Useful boundaries for most agents are context retrieval, prompt construction, model inference, output parsing, and action selection. Each stage has a narrow contract: inputs, outputs, and the error conditions it signals. Deploy each stage on [Amazon Bedrock AgentCore Runtime](#) with its own error handling and fallback behavior. Without this decomposition, all errors appear as generic reasoning failures, making debugging difficult. Clear stage boundaries enable precise error identification and faster resolution.

Tiering is where the stages earn their modularity. For context retrieval, primary tier uses [Amazon Bedrock Knowledge Bases](#) for semantic search, with fallback to simpler retrieval methods when the primary is unavailable. For model inference, implement model tier fallback using [Bedrock cross-region inference](#) for availability, substituting alternative models when the primary is degraded. For multimodal agents, [Amazon Bedrock Data Automation](#) preprocesses documents, images, audio, and video as a distinct reasoning stage before text-based reasoning, with independent fallbacks per modality.

Track per-stage error rates, latency, and fallback activation frequency through [Amazon Bedrock AgentCore Observability](#). Configure alarms that trigger automatic cutoffs when stage health degrades. The cutoff activates the fallback immediately rather than waiting for the next failed invocation. Log the retrieval tier and model tier used in each invocation so you can see, months later, which tier produced the answer and whether the fallback path is being taken more often than expected.

Implementation steps

1. **Decompose the reasoning pipeline into distinct stages:** Define explicit input/output schemas and deploy each stage on [Amazon Bedrock AgentCore Runtime](#).
2. **Implement automatic cutoffs between stages:** Activate stage-specific fallbacks when error rates exceed thresholds.
3. **Build tiered context retrieval:** Use [Amazon Bedrock Knowledge Bases](#) as primary with progressively simpler fallbacks.
4. **Implement model tier fallback:** Use [Bedrock cross-region inference](#) for availability during primary model degradation.

5. **Monitor per-stage health:** Track error rates, latency, and fallback activation through [Amazon Bedrock AgentCore Observability](#) with alarms that trigger automatic cutoffs.

Resources

Related best practices:

- [AGENTREL01-BP02 Establish modular, fault-isolated layers](#)
- [AGENTREL05-BP02 Facilitate reliable adaptation through evaluation-driven improvement cycles](#)
- [AGENTREL05-BP03 Ground agent cognition in real information](#)

Related documents:

- [Amazon Bedrock Knowledge Bases](#)
- [Amazon Bedrock Data Automation](#)
- [Amazon Bedrock cross-region inference](#)
- [Amazon Bedrock AgentCore Observability](#)
- [AWS fail-fast pattern](#)
- [Strands Agents Agent Loop](#)

Related videos:

- [AWS re:Invent 2024 - Using Strands Agents to build autonomous, self-improving AI agents \(AIM426\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Runtime tutorials](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)

AGENTREL05-BP02 Facilitate reliable adaptation through evaluation-driven improvement cycles

Agents degrade quietly when no one is watching, and runtime self-modification based on noisy feedback makes things worse. Structured feedback collection with offline evaluation and validated deployments keeps adaptation reliable because every change is measured before it reaches users.

Desired outcome:

- You collect action-level, task-level, and session-level feedback signals on every agent interaction.
- You run automated and LLM-as-a-judge evaluations periodically, comparing current behavior against golden-path examples.
- You validate prompt and configuration changes offline before deploying through gradual rollout.

Common anti-patterns:

- Deploying agents without feedback collection, missing the chance to identify systematic errors.
- Applying automated behavioral changes at runtime without offline validation, risking regression from noisy feedback.
- Skipping monitoring of the feedback loop itself, so silent pipeline failures block adaptation from happening.

Benefits of establishing this best practice:

- Task execution quality improves steadily through structured feedback collection and validated adjustments.
- Systematic errors get identified and corrected faster because automated analysis catches patterns humans miss.
- Manual intervention drops because evaluation-driven prompt optimization with controlled rollout replaces manual tuning.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Feedback is only useful at the granularity you collect it. Three tiers cover most of the signal. Action-level captures whether a tool call succeeded, task-level captures whether the agent completed the task correctly, and session-level captures whether the interaction achieved the user's goal. Action-level feedback tends to come from automated validators that compare outputs against expected schemas. Task-level feedback can be automated for deterministic success criteria and needs LLM-as-a-judge for subjective quality dimensions. Session-level feedback usually comes from users, either directly or through behavioral signals like follow-up questions.

[Amazon Bedrock AgentCore Evaluations](#) runs the periodic quality assessments against representative task sets, comparing outputs against golden-path examples and flagging regressions. Store evaluation results alongside task records so the agent's performance over time becomes a labeled dataset you can query. When evaluations indicate systematic degradation, that is the signal to trigger an offline prompt optimization workflow, test alternative formulations against evaluation benchmarks and deploy the highest-performing version through gradual rollout.

The discipline that keeps this reliable is validated before deployed, not modified at runtime. Runtime self-modification is tempting because it produces faster feedback, but noisy feedback can push agents into worse behavior. The scope of impact of a bad auto-update is the entire production fleet. Offline validation with gradual rollout keeps improvements under control. Monitor feedback loop health through [Amazon Bedrock AgentCore Observability](#). Track collection rates, processing latency, and evaluation frequency, with alarms when pipeline failures block the improvement cycle from operating.

Implementation steps

1. **Implement multi-tier feedback collection:** Capture action-level, task-level, and session-level signals for every interaction.
2. **Deploy automated outcome validators for deterministic criteria:** Compare outputs against expected schemas where the success criteria are unambiguous.
3. **Use AgentCore Evaluations with LLM-as-a-judge for subjective quality:** Run [Amazon Bedrock AgentCore Evaluations](#) on a periodic schedule against golden-path examples.
4. **Trigger offline prompt optimization when evaluations show degradation:** Validate candidates against benchmarks offline, then deploy through gradual rollout rather than runtime self-modification.

5. **Monitor feedback loop health:** Track collection rates, processing latency, and evaluation frequency through [Amazon Bedrock AgentCore Observability](#) with alarms for pipeline failures.

Resources

Related best practices:

- [AGENTREL02-BP03 Implement behavioral anomaly detection and monitoring](#)
- [AGENTREL05-BP01 Design modular, fault-tolerant agentic reasoning components](#)
- [AGENTREL05-BP03 Ground agent cognition in real information](#)

Related documents:

- [Amazon Bedrock AgentCore Evaluations](#)
- [Evaluate models in Amazon Bedrock](#)
- [Build reliable AI agents with Amazon Bedrock AgentCore Evaluations](#)
- [Evaluating AI agents: Real-world lessons from building agentic systems at Amazon](#)

Related videos:

- [AWS re:Invent 2024 - Using Strands Agents to build autonomous, self-improving AI agents \(AIM426\)](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)

AGENTREL05-BP03 Ground agent cognition in real information

Training data has a cutoff and an agent reasoning only from model knowledge can hallucinate about the present. Retrieval-augmented generation grounds each answer in current, domain-specific information and reduces hallucination rates as a byproduct.

Desired outcome:

- You have retrieval pipelines that ground agent reasoning in current, domain-specific information.
- You validate knowledge freshness and flag content that exceeds staleness thresholds.
- You handle retrieval failures gracefully, letting agents continue with model knowledge while communicating the uncertainty.

Common anti-patterns:

- Relying only on model training data for domain-specific knowledge, producing outputs that may be outdated or inaccurate.
- Running retrieval without freshness validation, causing agents to reason from stale data.
- Treating retrieval as a hard dependency, so retrieval failures cascade into agent failures.

Benefits of establishing this best practice:

- Hallucination rates drop because reasoning is grounded in retrieved factual information.
- Factual accuracy improves through access to current, domain-specific knowledge.
- Reliability holds as the operational environment evolves through knowledge base updates.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock Knowledge Bases](#) handles the mechanics of RAG, document ingestion, chunking, embedding, and vector storage, so most of the setup is configuration rather than infrastructure. Configure data sources that reflect the agent's domain and set up automated synchronization to keep content current. S3 event notifications trigger sync operations when source documents are updated, and the Knowledge Bases direct ingestion API handles programmatic content. Chunking strategy matters. Smaller chunks produce precise factual retrieval, while larger chunks produce better contextual understanding. Reranking models re-score retrieved passages for higher-quality context.

A knowledge base populated at launch and never refreshed becomes a source of wrong answers over time. Track ingestion timestamps and flag content that exceeds staleness thresholds before it is served. For information that requires real-time accuracy (prices, inventory, and system status), caches are not sufficient. Implement tool functions that agents invoke to retrieve data from

authoritative sources through [Amazon Bedrock AgentCore Gateway](#), and treat the authoritative source as the single source of truth.

[Amazon Bedrock Data Automation](#) extracts structured data from documents, forms, and tables, so agents reason over extracted content rather than raw images. Retrieved context quality assessment filters low-relevance results and deduplicates redundant passages before injection into prompts. Otherwise the context window fills with noise that drowns out the signal. Handle retrieval failures by allowing the agent to continue with model knowledge while communicating uncertainty about information currency. A transparent "I'm working from general knowledge rather than current data" beats silent reliance on training data.

Implementation steps

1. **Configure Amazon Bedrock Knowledge Bases with automated synchronization:** Set up [Amazon Bedrock Knowledge Bases](#) with domain-appropriate data sources and sync pipelines triggered by source changes.
2. **Implement knowledge freshness validation:** Track ingestion timestamps and flag stale content before it is served.
3. **Use Knowledge Bases reranking:** Re-score retrieved passages for higher-quality context injection.
4. **Implement real-time data retrieval tools through AgentCore Gateway:** Use [Amazon Bedrock AgentCore Gateway](#) tool functions for information that requires current accuracy.
5. **Handle retrieval failures gracefully:** Allow agents to continue with model knowledge while communicating uncertainty about information currency.

Resources

Related best practices:

- [AGENTREL03-BP01 Design an information classification model to identify short-term and long-term memories](#)
- [AGENTREL05-BP01 Design modular, fault-tolerant agentic reasoning components](#)
- [AGENTREL05-BP02 Facilitate reliable adaptation through evaluation-driven improvement cycles](#)

Related documents:

- [Amazon Bedrock Knowledge Bases](#)

- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock Data Automation](#)

Related videos:

- [AWS re:Invent 2024 - Advanced agentic RAG Systems: Deep dive with Bedrock \(AIM425\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agent-samples - Knowledge Base integration](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)

Legacy system integration

Agents that integrate with existing systems through adapter patterns and abstraction interfaces preserve legacy system stability while enabling automation of established workflows. These systems might not be optimized for agent interactions and might require support for protocols such as MCP or A2A. How do agents integrate effectively without impacting the reliability of established processes?

AGENTREL06: How do agents integrate effectively with existing systems without impacting the reliability of established processes?

Capability intent

- Legacy systems are reached through adapter interfaces that expose agent-native tool contracts, so agents don't need to understand the protocols behind them.
- Legacy systems are protected from agent-native invocation patterns through rate limiting and access control applied at the adapter layer.

- Every legacy dependency has a defined fallback path, so legacy outages cause reduced capability rather than complete agent failure.
- Agent operations with side effects are idempotent, so retry-based recovery is safe and doesn't produce duplicate transactions or inconsistent state.
- Operators can disable and re-enable individual capabilities at runtime without redeployment, and each capability has a tested fallback behavior that activates when it is toggled off.
- Resilience mechanisms are exercised regularly through fault injection and game days, so gaps are discovered in testing rather than during production incidents.

Maturity levels

These levels summarize what each stage of maturity looks like for legacy system integration as a whole.

Level	Name	What it looks like
1	Initial	Agents call legacy systems directly without adapter layers, so legacy error codes and protocol quirks leak into agent logic. There are no explicit fallbacks, so legacy outages cascade into complete agent failure. Retries are applied without idempotency, which can produce duplicate transactions, and disabling a problematic capability requires redeployment. Resilience is assumed rather than tested.
2	Emerging	Integration adapters are introduced for the most critical legacy systems, exposing tool interface

Level	Name	What it looks like
		s that translate to legacy protocols internally. One-time fallbacks are added after the first outages reveal the need, and retries start using idempotency keys for a small set of high-risk operations. Teams occasionally run manual failure tests before major releases. Telemetry on adapter health is partial and varies by system.

Level	Name	What it looks like
3	Defined	Adapters are registered in Amazon Bedrock AgentCore Gateway with a canonical error taxonomy, so agents handle legacy failures consistently. Each legacy dependency has a matched fallback strategy: cache-based for reference data, queue-based for transactional operations through Amazon SQS , and graceful degradation for real-time data. Idempotency is implemented through Amazon DynamoDB conditional writes and TTL-based expiration, and capability toggling through Amazon Bedrock AgentCore Policy . Cedar policies allow runtime control. AWS Fault Injection Service experiments run on a defined schedule.

Level	Name	What it looks like
4	Proactive	Legacy integration health is monitored through Amazon Bedrock AgentCore Observability with alarms that trigger automatic cutoffs using AWS Lambda circuit breaker logic, and recovery probes re-enable access when systems recover. Idempotency keys propagate through multi-step workflows and are passed to external systems that support native idempotency. Fallback behaviors are tested alongside primary implementations, and fallback activation rates feed a continuous prioritization of legacy reliability investment. FIS experiments are part of CI/CD, and quarterly game days validate runbooks under realistic conditions.

Level	Name	What it looks like
5	Optimized	Adapter patterns, fallback strategies, and idempotency mechanisms are standardized across the organization and continuously refined based on operational data. Capability toggles, fallback activation rates, and legacy health trends are integrated into decision-making for legacy modernization investment. Self-healing behavior (automatic cutoff, graceful degradation, and recovery) is the default, and game days rehearse multi-system failures rather than single-service outages. The organization publishes internal patterns for agent-to-legacy integration and shares benchmarks across teams.

Common issues to watch for

- Agents couple directly to legacy interfaces without adapter layers, so legacy protocol changes ripple through agent code and every agent handles legacy error codes in its own way.
- Legacy dependencies are assumed to have the reliability of cloud-based services, so no fallback paths exist and a single legacy outage causes complete agent failure.
- Retries are applied without idempotency guarantees, so recovery from transient failures creates duplicate transactions, corrupted state, or silent inconsistency.
- Disabling a problematic capability requires code changes and redeployment, which extends time-to-remediation during incidents and discourages cutting off misbehaving features.

- Automatic cutoffs open on failure but never close after recovery because no probe re-enables access, leaving capability degraded long after the legacy system is healthy again.
- Resilience mechanisms are never exercised, so fallback paths, runbooks, and alerting are only tested for the first time during a real incident.

Best practices

- [AGENTREL06-BP01 Develop agent-based integrations with existing or legacy systems](#)
- [AGENTREL06-BP02 Establish fallback mechanisms for legacy system degradation](#)
- [AGENTREL06-BP03 Regularly test degraded system performance](#)
- [AGENTREL06-BP04 Implement idempotent task execution patterns](#)
- [AGENTREL06-BP05 Implement dynamic capability toggling](#)

AGENTREL06-BP01 Develop agent-based integrations with existing or legacy systems

Legacy systems expose interfaces built for synchronous, deterministic callers, while agents are asynchronous and probabilistic. Adapter layers translate between the two worlds so agents can use existing capabilities without the legacy side needing to change.

Desired outcome:

- You have integration adapters that expose MCP tool interfaces designed for agent consumption and translate to legacy protocols internally.
- You enforce canonical error handling that maps legacy error codes into types agents can interpret uniformly.
- You rate-limit at the adapter layer so agent-driven invocation speeds don't overwhelm legacy systems.

Common anti-patterns:

- Requiring legacy system modifications to support agent integration, adding deployment risk and organizational friction.
- Coupling agents directly to legacy interfaces without adapters, exposing agents to legacy-specific complexity.

- Skipping rate limiting on legacy system calls, producing overload that degrades performance for every consumer.

Benefits of establishing this best practice:

- Legacy system stability is preserved through adapter layers that shield it from agent interaction patterns.
- Agents and legacy systems evolve on independent schedules through abstraction interfaces.
- Agent adoption accelerates because integration doesn't require legacy modifications.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Adapters resolve the impedance mismatch between agent and legacy. Expose the legacy capability as an MCP tool with a schema an agent can reason about. Internally, translate to whatever the legacy system speaks: SOAP, screen scraping, database queries, or batch files. [Amazon Bedrock AgentCore Gateway](#) registers adapters as discoverable tools with consistent authentication policies. Agents then invoke legacy capabilities through the same tool-call pattern they use for cloud-based capabilities. The [blog on uniting MCP servers through AgentCore Gateway](#) covers the pattern for unifying multiple adapters behind a single gateway interface.

Error mapping makes the adapter actually useful. Legacy systems emit error codes specific to their architecture, and exposing those codes directly forces every agent to understand every legacy system. A canonical error taxonomy (connection timeout, authentication failure, rate limit exceeded, and system unavailable) lets agents apply consistent handling logic without knowing the specifics of each legacy system. The translation from legacy code to canonical type happens inside the adapter.

Rate limiting helps protect the legacy side. Legacy systems were sized for human-paced traffic, not for an agent that can invoke tools as fast as the LLM generates tool calls. Application-level rate limiting in the adapter layer throttles agent-driven invocation speeds to something the legacy system can handle. Combine this with [Amazon Bedrock AgentCore Policy](#) Cedar policies that restrict which agents can invoke legacy adapters. Monitor adapter health through [Amazon Bedrock AgentCore Observability](#) to detect legacy integration reliability issues before they become incidents.

Implementation steps

1. **Implement integration adapters exposing MCP tool interfaces:** Translate to legacy protocols internally so agents see a uniform tool-call interface.
2. **Register adapters in AgentCore Gateway:** Expose adapters through [Amazon Bedrock AgentCore Gateway](#) with authentication and discovery for agent invocation.
3. **Implement canonical error mapping:** Translate legacy error codes into a consistent taxonomy agents can handle uniformly.
4. **Enforce access control and rate limiting:** Use [Amazon Bedrock AgentCore Policy](#) for access control and implement application-level rate limiting in the adapter to protect legacy systems.
5. **Monitor adapter health through AgentCore Observability:** Detect legacy integration reliability issues through [Amazon Bedrock AgentCore Observability](#).

Resources

Related best practices:

- [AGENTREL06-BP02 Establish fallback mechanisms for legacy system degradation](#)
- [AGENTREL06-BP03 Regularly test degraded system performance](#)
- [AGENTREL06-BP04 Implement idempotent task execution patterns](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway](#)
- [Transform your MCP architecture: Unite MCP servers through AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Policy](#)

Related videos:

- [AWS re:Invent 2024 - Adding agentic AI to legacy apps with AgentCore \(MAM345\)](#)

Related services:

- [Amazon Bedrock AgentCore](#)

AGENTREL06-BP02 Establish fallback mechanisms for legacy system degradation

Legacy systems carry lower availability SLAs than cloud-based services, and their failure modes are often unpredictable. Health-aware fallback paths, caches for reference data, queues for transactions, graceful degradation for real-time, keep agent workflows running through legacy outages.

Desired outcome:

- You have health monitoring on every legacy integration with alarms that trigger fallback activation.
- You have cache-based fallbacks for reference data and queue-based fallbacks for transactional operations.
- You recover automatically when legacy systems return, with periodic probes deactivating the cutoff.

Common anti-patterns:

- Assuming legacy systems match cloud-based reliability, without implementing fallbacks for their actual SLAs.
- Deploying fallbacks that silently return stale or incorrect data without informing agents or users.
- Skipping legacy health monitoring, so outages become visible only when agent tasks fail.

Benefits of establishing this best practice:

- Agent functionality stays available during legacy outages through pre-defined fallback paths.
- Proactive fallback activation through health monitoring replaces reactive failure detection.
- Users and downstream systems see transparent indications of degraded capability.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Health monitoring is the prerequisite for any automatic fallback. Without a health signal, the only way to know the legacy system is down is to watch user-visible failures accumulate. Periodic probes

through [Amazon Bedrock AgentCore Observability](#) and Amazon CloudWatch check endpoint availability and response time. Alarms on those probes trigger fallback activation before the first user-facing failure happens.

Fallback shape depends on the operation type. For reference data, product catalogs, configuration values, mostly-static lookups, cache-based fallbacks serve recently retrieved data during outages. The accuracy cost is low because the data doesn't change often. For transactional operations, queue-based fallbacks buffer requests for replay when the system recovers, preserving the intent of each operation without attempting it against an unreachable system. For real-time data that can't be cached or queued, live prices, current inventory, instantaneous system state, graceful degradation means informing the user the information is temporarily unavailable rather than returning a plausible-but-wrong answer.

Automatic cutoffs need to unwind themselves, or every outage turns into a permanent downgrade. Use CloudWatch alarms to detect when error rates cross a threshold and trigger automated responses, updating [Amazon Bedrock AgentCore Policy](#) Cedar policies to deny tool access or activating Lambda-based circuit breaker logic. Configure periodic probes that test availability and re-enable access when the system recovers. Monitor fallback activation frequency through AgentCore Observability to identify legacy systems causing disproportionate reliability issues. Those systems are the candidates for modernization investment.

Implementation steps

1. **Implement health monitoring for each legacy integration:** Use [Amazon Bedrock AgentCore Observability](#) and Amazon CloudWatch probes to check endpoint availability and response time.
2. **Configure alarms that trigger fallback activation:** Alarm on health degradation so fallbacks activate before user-visible failures accumulate.
3. **Implement operation-appropriate fallbacks:** Cache-based fallbacks for reference data, queue-based fallbacks for transactional operations, and graceful degradation messages for real-time data.
4. **Deploy automatic cutoffs with recovery detection:** Use CloudWatch alarms to trigger [Amazon Bedrock AgentCore Policy](#) updates or circuit breaker logic, and run periodic probes to re-enable access when the system recovers.
5. **Monitor fallback activation frequency:** Use AgentCore Observability to identify legacy systems that consistently degrade, so modernization effort can be prioritized.

Resources

Related best practices:

- [AGENTREL04-BP03 Implement fallback mechanisms and graceful degradation for collaborative workflows](#)
- [AGENTREL06-BP01 Develop agent-based integrations with existing or legacy systems](#)
- [AGENTREL06-BP03 Regularly test degraded system performance](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock AgentCore Policy](#)
- [Build resilient generative AI agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTREL06-BP03 Regularly test degraded system performance

Resilience claims that have never been tested under real failure conditions are just aspirations. Regular chaos engineering, fault injection, and load testing under constrained resources reveal the gaps in fallback coverage while the environment is safe to break.

Desired outcome:

- You have experiment templates for the failure scenarios most likely to affect agent reliability.
- You have documented acceptance criteria for each scenario, covering expected fallback activation, acceptable degradation, and recovery time.
- You run fault-injection experiments at least monthly and track findings through a resilience improvement backlog.

Common anti-patterns:

- Testing only happy-path scenarios, discovering resilience gaps only during production incidents.
- Running degraded testing infrequently, allowing resilience regressions to accumulate between cycles.
- Testing individual components in isolation without full-workflow failure scenarios.

Benefits of establishing this best practice:

- Resilience gaps get discovered before they reach production incidents.
- Fallback mechanisms are validated against real failure conditions rather than hypothetical ones.
- Resilience assurance keeps pace with system evolution through regular testing cycles.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

[AWS Fault Injection Service](#) is the managed way to inject controlled failures into agent infrastructure, throttling, node failures, network partitions, and observe system behavior. The experiment is only as useful as the acceptance criteria it is compared against, so every scenario needs documented expectations. Define which fallback should activate, what capability degradation is acceptable, and how long recovery should take. Running the experiment without criteria gives you an interesting demo. Running it with criteria gives you a regression test.

Monthly is the minimum frequency that keeps resilience regressions from accumulating between cycles. Integrating degraded testing into CI/CD blocks production deployment when tests fail, which is where resilience assurance actually gets enforced. Run the experiments in non-production environments scoped tightly enough that you are not causing incidents you were trying to prevent.

Game days extend the practice into operational readiness. Quarterly game days where the operations team deliberately induces failures in production-like environments validate more than technical fallback mechanisms. They also exercise operational runbooks, alerting configuration, and team response under time pressure. The findings get documented in a resilience improvement backlog and tracked to remediation. [Amazon Bedrock AgentCore Observability](#) monitors system behavior during tests and confirms that degradation detection triggers correctly.

Implementation steps

1. **Create FIS experiment templates for high-risk scenarios:** Build [AWS Fault Injection Service](#) templates for the failure modes most likely to affect agent reliability, scoped to non-production environments.
2. **Define acceptance criteria per scenario:** Document expected fallback activation, acceptable degradation, and recovery time so each experiment has a pass/fail bar.
3. **Integrate degraded testing into CI/CD:** Block production deployment when tests fail so resilience assurance is enforced rather than aspirational.
4. **Run experiments at least monthly:** Schedule FIS experiments on a regular cadence and track results to detect resilience regressions.
5. **Run quarterly game days:** Exercise operational runbooks, alerting, and team response procedures under controlled but realistic failure conditions.

Resources

Related best practices:

- [AGENTRELO6-BP01 Develop agent-based integrations with existing or legacy systems](#)
- [AGENTRELO6-BP02 Establish fallback mechanisms for legacy system degradation](#)
- [AGENTRELO6-BP04 Implement idempotent task execution patterns](#)

Related documents:

- [AWS Fault Injection Service](#)
- [Build resilient generative AI agents](#)
- [Amazon Bedrock AgentCore Observability](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Runtime tutorials](#)

Related services:

- [AWS Fault Injection Service](#)

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTREL06-BP04 Implement idempotent task execution patterns

Retry is the most common recovery mechanism, and without idempotency it can produce duplicate side effects. Deterministic idempotency keys and conditional writes let operations be retried safely without producing duplicate side effects.

Desired outcome:

- You generate deterministic idempotency keys from operation inputs, so retries of the same logical operation always produce the same key.
- You check for an existing result before executing any side-effectful operation and return the cached result if the operation already succeeded.
- You propagate idempotency keys through multi-step workflows and to external systems that support built-in idempotency.

Common anti-patterns:

- Implementing retries without idempotency guarantees, producing duplicate side effects when operations are retried after partial completion.
- Using non-deterministic identifiers for idempotency keys, so retries of the same operation generate different keys and defeat the guarantee.
- Failing to propagate idempotency keys through multi-step workflows, allowing duplicate effects at steps that don't receive the original key.

Benefits of establishing this best practice:

- Retry-based recovery from transient failures becomes safe without duplicate side effects.
- Error handling simplifies because first-attempt and retry executions look identical.
- Multi-step workflows recover reliably because idempotency keys flow through every step.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Idempotency key generation is where most teams go wrong. Non-deterministic keys (timestamps and UUIDs generated at retry time) defeat the mechanism because the retry computes a different key from the original attempt. Deterministic keys derived from operation inputs work. Hash the workflow ID, task type, and request body, and the same logical operation always produces the same key. This is the single most important detail to get right because everything else depends on it.

Once the key is stable, the pre-execution check becomes trivial. Before executing an operation with side effects, query the idempotency store for an existing result keyed on the same identifier. If you find one and it succeeded, return the cached result without re-executing. If you don't, run the operation and record the result. [Amazon DynamoDB](#) with conditional writes makes this safe under concurrency. Two parallel retries can't both create a record, and TTL-based expiration keeps the store from growing without bound while maintaining the guarantee within the expected retry window.

Propagation extends the guarantee across workflows. When an orchestrator delegates a subtask, include the parent workflow's key or a deterministic derivative so downstream agents maintain idempotency through every step. For operations interacting with external systems that support built-in idempotency mechanisms, pass the agent's key to the external system. Monitor idempotency store health through [Amazon Bedrock AgentCore Observability](#), tracking cache hit rates. Retries that return cached results are the signal the mechanism is working as intended.

Implementation steps

1. **Design deterministic idempotency keys:** Derive keys from operation inputs through consistent hashing so retries generate the same key as the original attempt.
2. **Implement idempotency checking as a pre-execution step:** Use [Amazon DynamoDB](#) with conditional writes and TTL-based expiration to store idempotency records safely.
3. **Propagate idempotency keys through multi-step workflows:** Maintain idempotency across all steps by passing the parent workflow's key or a deterministic derivative to downstream steps.
4. **Pass keys to external systems with built-in idempotency:** Prevent duplicate processing at the external system level by forwarding the agent's key.
5. **Monitor cache hit rates:** Use [Amazon Bedrock AgentCore Observability](#) to watch for retries that return cached results, confirming the mechanism is operating.

Resources

Related best practices:

- [AGENTREL06-BP01 Develop agent-based integrations with existing or legacy systems](#)
- [AGENTREL06-BP03 Regularly test degraded system performance](#)
- [AGENTREL06-BP05 Implement dynamic capability toggling](#)

Related documents:

- [Amazon DynamoDB](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Build resilient generative AI agents](#)

Related services:

- [Amazon DynamoDB](#)
- [Amazon Bedrock AgentCore](#)

AGENTREL06-BP05 Implement dynamic capability toggling

Taking an entire agent offline to fix one misbehaving feature is disproportionate. Feature flags at the gateway boundary let operators disable the affected capability immediately, keeping the rest of the agent usable while fixes or investigations proceed.

Desired outcome:

- You have capability toggles that disable specific tools or features without redeploying the agent.
- You have fallback behaviors defined for each capability so agents continue operating when a feature is disabled.
- You monitor toggle state and fallback usage so capabilities exhibiting silent degradation surface through elevated fallback rates.

Common anti-patterns:

- Requiring redeployment to disable problematic capabilities, extending time to remediation.

- Implementing toggles without fallback behaviors, so disabling a capability triggers agent failures rather than reduced functionality.
- Omitting user-facing communication when features are unavailable, leaving users confused about current capability.

Benefits of establishing this best practice:

- Capability-specific issues get remediated quickly without full agent redeployment.
- Overall agent functionality survives capability toggles through graceful fallbacks.
- New capabilities can be rolled out gradually and rolled back immediately when problems appear.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Gateway-level toggling is the control point. [Amazon Bedrock AgentCore Policy](#) Cedar policies can enable or disable specific tool access for agents without redeployment, providing immediate control over which capabilities are available. Policy updates propagate quickly, so the toggle can be flipped while an incident is still active rather than waiting for a deployment window. For more granular runtime toggling within agent logic, use a managed configuration service with local caching and configurable polling intervals so agents respond to changes within seconds.

Fallback behaviors are what make toggling safe. Define what the agent does when a capability is disabled. Examples include semantic search falling back to keyword search, complex reasoning falling back to simpler rules, and real-time data falling back to cached values. Document fallback behaviors alongside capability definitions so operators know the impact of flipping each toggle. Treat fallback paths as first-class code and test them alongside the primary implementations so they actually work when you need them.

Monitoring makes the system self-aware. Track capability toggle state and fallback usage through [Amazon Bedrock AgentCore Observability](#). Alarms on fallback rates that exceed baseline signal capabilities experiencing issues even when not explicitly toggled off, giving operators early warning before users report problems. This turns the toggle mechanism from a manual lever into a proactive detection surface.

Implementation steps

1. **Implement capability toggling through AgentCore Policy:** Use Cedar policies in [Amazon Bedrock AgentCore Policy](#) to control tool access at the gateway boundary.
2. **Define and implement fallback behaviors for each capability:** Document the reduced-capability behavior that activates when a toggle is flipped.
3. **Test fallback paths alongside primary implementations:** Validate that fallbacks produce acceptable results, not just that they don't error.
4. **Monitor toggle state and fallback usage:** Track both through [Amazon Bedrock AgentCore Observability](#).
5. **Configure alarms on elevated fallback rates:** Detect capabilities experiencing issues before users report them.

Resources

Related best practices:

- [AGENTREL06-BP01 Develop agent-based integrations with existing or legacy systems](#)
- [AGENTREL06-BP02 Establish fallback mechanisms for legacy system degradation](#)
- [AGENTREL06-BP04 Implement idempotent task execution patterns](#)
- [AGENTREL08-BP01 Establish consistent configuration management practices](#)

Related documents:

- [Amazon Bedrock AgentCore Policy](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Secure AI agents with Policy in Amazon Bedrock AgentCore](#)
- [Amazon Bedrock AgentCore Observability](#)

Related services:

- [Amazon Bedrock AgentCore](#)

Agent monitoring, management and recovery

Agent systems that decompose workflows into recoverable stages, classify failures for targeted retry, and implement end-to-end distributed tracing recover smoothly from the failures that occur across their many components.

AGENTREL07: How do fault tolerant agent systems recover?

Capability intent

- Agent workflows are decomposed into stages with persisted outputs and explicit validation between them, so failures are contained to the affected stage rather than cascading through the whole workflow.
- Failures are classified before any recovery action is taken, so retries apply to transient errors, fallbacks apply to persistent ones, and only genuinely unrecoverable failures reach human attention.
- Retries use exponential backoff with jitter and a retry budget, so widespread upstream failures don't produce unbounded retry storms.
- Self-healing responses address common failure patterns directly (prompt refinement, tool substitution, context reconstruction) before escalating to fallbacks.
- End-to-end distributed tracing covers every agent invocation, with agent-specific annotations and framework-level reasoning steps captured in the same trace view.
- Traces, metrics, and logs are correlated in unified dashboards, so operators diagnose incidents from evidence rather than by pivoting between tools.

Maturity levels

These levels summarize what each stage of maturity looks like for agent monitoring, management and recovery as a whole.

Level	Name	What it looks like
1	Initial	Agent workflows run as monolithic processes, so any failure causes a complete

Level	Name	What it looks like
		restart and lost work. Retries are applied uniformly, including to non-retryable errors, and without exponential backoff or jitter. Tracing is limited to the application boundary, and incident responders reconstruct the execution path from logs. Recovery actions are manual.
2	Emerging	A few long-running workflows are split into stages, and retries use exponential backoff for a limited set of known-transient errors. Amazon CloudWatch alarms notify operators on elevated error rates. Tracing is added for critical paths but doesn't propagate across queue boundaries. Failure classification exists in one-time code rather than a shared library.

Level	Name	What it looks like
3	Defined	Workflows are orchestrated in AWS Step Functions with state persistence at every transition and redrive enabled for incremental recovery. Failure classification is standardized in a shared library, and retries use exponential backoff with full jitter. Amazon Bedrock AgentCore Observability captures end-to-end traces across agent components, and Amazon Bedrock AgentCore Policy enforces retry budgets. Stage-level validation helps prevent errors from propagating through later stages.

Level	Name	What it looks like
4	Proactive	Self-healing responses address common failure patterns directly (prompt refinement, tool substitution, context reconstruction) before fallbacks engage. Fallback strategies route persistent failures to degraded responses, cached answers, or human review queues rather than returning errors. Distributed tracing includes agent-specific annotations and Strands Agents reasoning steps, and trace context propagates through every communication path including asynchronous messages. Amazon Bedrock AgentCore Evaluations validate that recovered outputs match pre-failure quality baselines.

Level	Name	What it looks like
5	Optimized	Recovery strategies are continuously refined based on recovery metrics and trace analysis. Fallback paths are tested alongside primary implementations, and stage decomposition is driven by observed failure and timeout patterns. Trace-based alerting fires on anomalies before they become incidents, and correlated dashboards turn traces, metrics, and logs into a single investigative surface. The organization publishes internal recovery patterns and shares lessons across teams.

Common issues to watch for

- Long-running workflows run as monolithic processes, so a failure late in the workflow loses all progress and forces complete restart.
- Retry logic is applied uniformly to every failure, retrying non-retryable errors that will never succeed and delaying the human intervention that would resolve them.
- Retries run at fixed intervals without exponential backoff or jitter, producing thundering-herd effects that amplify incidents rather than recovering from them.
- Only retry-based recovery is implemented, so persistent failures after retries become user-visible errors instead of degrading through a fallback.
- Tracing stops at the application boundary or at queue boundaries, so debugging failures in multi-agent systems depends on manual log correlation across services.
- Traces are captured without agent-specific annotations, so trace queries can't be filtered down to the specific agent, task type, or model that failed.

Best practices

- [AGENTREL07-BP01 Design workflows in stages with incremental recovery](#)
- [AGENTREL07-BP02 Enable automatic recovery from agent execution failures](#)
- [AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery](#)

AGENTREL07-BP01 Design workflows in stages with incremental recovery

Monolithic workflows lose everything on a single failure. Explicit stage boundaries with persisted outputs contain failures to the affected stage and let recovery start from the last completed checkpoint rather than the beginning.

Desired outcome:

- You have workflows decomposed into discrete stages at natural checkpoints where completed work has independent value.
- You persist stage outputs durably so recovery resumes from the last completed stage.
- You validate stage outputs before advancing so errors don't propagate silently through subsequent stages.

Common anti-patterns:

- Running workflows as monolithic processes without stage boundaries, so any failure forces a complete restart.
- Defining stages at too coarse a granularity, losing large amounts of work within a stage when it fails.
- Skipping stage output validation, allowing errors to propagate through subsequent stages.

Benefits of establishing this best practice:

- Work loss stays minimal because recovery resumes from the last completed stage.
- Recovery is faster because redundant recomputation of completed stages is avoided.
- Stage boundaries contain failures, improving error isolation.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[AWS Step Functions](#) persists execution state at every state transition and enables recovery from the last completed step rather than restarting entirely. The built-in retry with exponential backoff handles transient errors within a step, and the redrive capability restarts the workflow from the point of failure without re-executing completed steps. This combination, persistence plus selective retry plus redrive, is what gives incremental recovery its teeth. [Amazon Bedrock AgentCore Runtime](#) provides the execution surface for individual agent steps within the workflow.

Place stage boundaries where completed work has independent value. A parsed document, a validated query, and a retrieved and summarized context are all natural boundaries. If a stage produces a half-built artifact that is useless on its own, the boundary is in the wrong place.

Quality protection follows stage design. Stage output validation between stages, checking schema conformance and quality thresholds before advancing, keeps errors from propagating. [Amazon Bedrock AgentCore Evaluations](#) can verify that recovered workflow outputs match pre-failure quality baselines, so incremental recovery doesn't silently degrade quality. Stage-level timeouts prevent stuck stages from blocking progress indefinitely. Stage-level metrics through [Amazon Bedrock AgentCore Observability](#), including success rate, execution time, and timeout frequency, identify stages that need optimization.

Implementation steps

1. **Decompose workflows into discrete stages:** Place boundaries at natural checkpoints where completed work has independent value.
2. **Implement with Step Functions for durable state:** Use [AWS Step Functions](#) with built-in retry and exponential backoff per step.
3. **Configure redrive for recovery from the point of failure:** Restart failed workflows without re-executing completed steps.
4. **Implement stage output validation:** Check schema conformance and quality thresholds between stages so errors don't propagate.
5. **Configure stage-level timeouts with recovery paths:** Handle stages that fail after exhausting retries.
6. **Monitor stage-level metrics:** Use [Amazon Bedrock AgentCore Observability](#) to find stages that need optimization.

Resources

Related best practices:

- [AGENTREL03-BP03 Implement comprehensive state management and checkpoint-based recovery](#)
- [AGENTREL07-BP02 Enable automatic recovery from agent execution failures](#)
- [AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery](#)

Related documents:

- [AWS Step Functions](#)
- [Amazon Bedrock AgentCore Runtime](#)
- [Build resilient generative AI agents](#)
- [Planning for failure: How to make generative AI workloads more resilient](#)

Related videos:

- [AWS 2025 - AgentCore now GA: From Prototype to Production](#)

Related services:

- [AWS Step Functions](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTREL07-BP02 Enable automatic recovery from agent execution failures

Uniform retry wastes effort on non-retryable failures and creates thundering herds on transient ones. Classifying failures by type and applying targeted strategies, retry for transient errors, fallback for persistent ones, escalation for unrecoverable ones, keeps availability high without manual intervention.

Desired outcome:

- You classify agent failures into retryable and non-retryable categories at the point of failure.
- You use exponential backoff with full jitter on retryable failures, with failure-type-specific retry counts.
- You enforce retry budgets to help prevent retry storms, and escalate to fallback or human review when retries are exhausted.

Common anti-patterns:

- Applying uniform retry logic to every failure type, retrying non-retryable failures that will never succeed.
- Retrying at fixed intervals without exponential backoff and jitter, producing thundering-herd effects during recovery.
- Implementing only retry-based recovery without fallback strategies for failures that persist after retries.

Benefits of establishing this best practice:

- Availability stays high because transient failures resolve automatically.
- Resource utilization is efficient because non-retryable failures don't consume retry budget.
- Persistent failures get handled gracefully through fallback strategies when retries are exhausted.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Failure classification comes first because the recovery strategy depends on the category. Retryable failures include transient infrastructure errors, LLM throttling, and temporary unavailability, conditions that a short wait and a retry are likely to resolve. Non-retryable failures include authentication errors, invalid inputs, and permission denials, conditions where the same call will fail the same way no matter how many times you retry. Collapsing the two into a single category means either retrying things that will never succeed or giving up on things that would have worked.

Backoff strategy matters as much as classification. Exponential backoff with full jitter spreads retry attempts across time, avoiding the thundering herd that fixed-interval retries produce during widespread failures. Failure-type-specific retry counts help too: aggressive retry for transient

errors, conservative retry with longer intervals for rate limiting. Enforce retry budgets at two levels. For each invocation, use [Amazon Bedrock AgentCore Policy](#) to cap retries within a single agent execution. This helps prevent runaway loops without external dependencies. Across invocations, implement a shared circuit-breaker backed by a low-latency store (e.g., DynamoDB atomic counters or ElastiCache) that tracks cumulative retry counts across concurrent executions. AgentCore Policy evaluates the circuit-breaker state as a policy input, rejecting new retries when the global budget is exhausted. This two-tier approach avoids the anti-pattern of assuming single-invocation limits provide system-wide protection.

Self-healing workflows extend retry into recovery. Common failure patterns have targeted remediations. Automatic prompt refinement handles LLM output validation failures, tool substitution handles tool call failures, and context reconstruction handles memory access failures. When retries are exhausted, transition to fallback strategies or human review rather than terminating. Log every recovery action through [Amazon Bedrock AgentCore Observability](#) with structured metadata including retry counts, strategy used, and outcome, so recovery effectiveness can be analyzed and tuned.

Implementation steps

1. **Implement failure classification:** Categorize agent failures into retryable and non-retryable types at the point of failure.
2. **Configure retry with exponential backoff and full jitter:** Use failure-type-specific retry counts so the strategy matches the failure mode.
3. **Enforce retry budgets through AgentCore Policy:** Use [Amazon Bedrock AgentCore Policy](#) to help prevent retry storms during widespread failures.
4. **Build self-healing workflows for common patterns:** Implement prompt refinement, tool substitution, and context reconstruction as targeted recovery paths.
5. **Log recovery actions for analysis:** Use [Amazon Bedrock AgentCore Observability](#) to capture retry counts, strategies, and outcomes.

Resources

Related best practices:

- [AGENTREL06-BP04 Implement idempotent task execution patterns](#)
- [AGENTREL07-BP01 Design workflows in stages with incremental recovery](#)

- [AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery](#)

Related documents:

- [Amazon Bedrock AgentCore Policy](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Build resilient generative AI agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)

AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery

Correlating logs across services by hand is slow, and during an incident slow is worse than expensive. Distributed tracing across all components with agent-specific annotations gives operators the full request path in one view and turns broad restarts into targeted recovery actions.

Desired outcome:

- You have distributed tracing across every agent component with agent-specific annotations.
- You propagate trace context through synchronous and asynchronous communication boundaries.
- You correlate traces, metrics, and logs in a unified view that surfaces root causes quickly.

Common anti-patterns:

- Tracing only at the application boundary without propagating context through internal service calls.
- Skipping correlation of traces with logs and metrics, reducing the risk of unified analysis during incidents.
- Omitting agent-specific annotations that make filtering by agent ID, task type, or model used possible.

Benefits of establishing this best practice:

- Root causes surface quickly because the request flow is visible end to end.
- Mean time to recovery drops because trace data drives targeted actions instead of broad restarts.
- Latency bottlenecks become visible, enabling proactive performance optimization.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock AgentCore Observability](#) captures the full execution path of each agent invocation through OpenTelemetry-compatible telemetry. Turn it on across every agent component, not only the outer boundary, so the trace actually spans the request end to end. Without component-level coverage, traces have gaps where invisible work happens, which is exactly where debugging slows down during an incident.

Annotations are what make traces searchable at the scale of real systems. Agent-specific tags, agent ID, task type, model ID, workflow ID, let you filter traces to a specific agent or failure scenario instead of grepping through an undifferentiated stream. Instrument Strands Agents framework-level traces to capture reasoning steps, tool invocations, and their outcomes in a unified trace view, because the agent's internal decisions are where most of the interesting signals live.

Context propagation is the detail that decides whether asynchronous paths are visible. For queue-based communication, propagate trace headers through message attributes so traces continue across queue boundaries. Without propagation, the trace ends at the producer and a new trace starts at the consumer, and the fact that they relate is lost. Create trace-based alerting through Amazon CloudWatch that correlates traces, metrics, and logs in a unified view, so a trace anomaly, a metric spike, and a log error appear together rather than separately.

Implementation steps

1. **Enable AgentCore Observability with OpenTelemetry tracing:** Turn on [Amazon Bedrock AgentCore Observability](#) across every agent component.
2. **Add agent-specific annotations:** Tag traces with agent ID, task type, and model ID so filtering during incidents is possible.
3. **Propagate trace context across boundaries:** Include synchronous and asynchronous paths, with message attributes for queue-based communication.

4. **Instrument Strands Agents framework-level traces:** Capture reasoning steps and tool invocations in the unified trace view.
5. **Create CloudWatch dashboards that correlate traces, metrics, and logs:** Build one unified view so incident response works on signal.

Resources

Related best practices:

- [AGENTREL07-BP01 Design workflows in stages with incremental recovery](#)
- [AGENTREL07-BP02 Enable automatic recovery from agent execution failures](#)
- [AGENTREL08-BP02 Implement agent tracing for telemetry throughout agent processing](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Build trustworthy AI agents with Amazon Bedrock AgentCore Observability](#)
- [Strands Agents Traces](#)

Related videos:

- [AWS 2025 - AgentCore Observability: Monitor and Debug with OpenTelemetry](#)
- [AWS re:Invent 2024 - Observability for Reliable Agentic AI with Strands & OpenTelemetry \(NTA406\)](#)
- [AWS re:Invent 2024 - Build observable AI agents with Strands, AgentCore, and Datadog \(AIM233\)](#)

Related workshops:

- [Diving Deep into Bedrock AgentCore - Observability](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Graceful degradation and configuration management

Agents that detect degradation accurately through thorough telemetry and maintain consistent configuration across all instances make informed decisions about when to activate graceful degradation. Consistent configuration management helps agents have reliable, up-to-date information about capabilities and limitations across the agent environment.

AGENTREL08: How do agents determine when and where graceful degradation is appropriate?

Capability intent

- Agent configuration lives in a central, versioned, schema-validated source, so every instance runs the same current settings and drift is detected automatically.
- Configuration changes roll out gradually with automatic rollback, and sensitive values are held in encrypted parameter storage with fine-grained access control.
- Stage-level telemetry covers every phase of agent processing (context retrieval, inference, tool execution, response generation), so degradation decisions are informed rather than reactive.
- Anomaly detection and composite alarms combine signals across stages into a single useful health state, triggering graceful degradation automatically.
- Resource isolation separates high-priority user-facing agents from background workloads, and contention is detected through composite scores before it causes failures.
- Memory utilization is tracked per tier (in-context, short-term session, long-term persistent) with automated responses for summarization, pruning, and consolidation before exhaustion produces silent failures.

Maturity levels

These levels summarize what each stage of maturity looks like for graceful degradation and configuration management as a whole.

Level	Name	What it looks like
1	Initial	Configuration is hardcoded in agent code, so every parameter change requires a redeployment. Telemetry exists only at the request boundary, which hides stage-level degradation. Agents share a single resource pool, so any workload spike degrades every tenant. Memory utilization is monitored only at the infrastructure level, and in-context exhaustion produces silent failures. Graceful degradation is activated manually, if at all.
2	Emerging	Configuration is centralized but managed manually, with one-time versioning and limited schema validation. Basic Amazon CloudWatch metrics capture inference latency and error rate, but stage-level telemetry is partial. Resource isolation is informal, with heavy workloads sometimes placed on separate infrastructure. Memory growth is tracked for a few known-leaky components, and degradati

Level	Name	What it looks like
		on responses are scripted but triggered by operators.
3	Defined	Runtime configuration is managed through a central service such as AWS AppConfig with JSON Schema validation and gradual rollout, and secrets live in AWS Secrets Manager or Parameter Store SecureString parameters . Amazon Bedrock AgentCore Observability captures stage-level telemetry, and Amazon Bedrock model invocation logging records per-call token counts and latency. Priority tiers have separate Amazon Bedrock AgentCore Runtime instances, and memory utilization is tracked per tier with alarms at 80% context-window utilization.

Level	Name	What it looks like
4	Proactive	CloudWatch anomaly detection establishes baselines automatically, and CloudWatch composite alarms combine multi-stage signals into automated graceful degradation. Resource contention is detected through composite scores that combine concurrency, token consumption, and queue depths, and mitigation is automated through Amazon Bedrock AgentCore Policy . Amazon Bedrock Provisioned Throughput gives latency-sensitive agents dedicated capacity. Memory growth is analyzed through CloudWatch Metric Math , and summarization, pruning, and consolidation responses are automated. Drift monitoring alerts on configuration fleets running mixed versions.

Level	Name	What it looks like
5	Optimized	Degradation policies and configuration patterns are continuously refined based on operational data. Composite health scores, anomaly thresholds, and automated mitigation rules are tuned from observed outcomes rather than intuition. Memory management responses are self-healing across all tiers, and resource allocation adjusts dynamically based on priority and observed demand. The organization publishes reusable telemetry and configuration patterns internally and shares benchmarks on graceful degradation effectiveness across teams.

Common issues to watch for

- Configuration is hardcoded or partially centralized, so parameter changes require redeployment and configuration drift between instances is detected only through downstream failures.
- Configuration changes are applied without validation or staged rollout, which lets misconfigured values reach production and degrade agent behavior instantly.
- Telemetry is captured only at the request boundary, so a rise in overall latency or error rate provides no signal about which processing stage is responsible.
- All agents share a single resource pool and upstream quota, so a high-volume background agent degrades the latency of user-facing agents and throttling cascades across workloads.
- Memory monitoring stops at infrastructure metrics, so in-context exhaustion and gradual memory leaks produce silent output degradation with no leading indicator.

- Static alarm thresholds either fire too often during routine traffic shifts or miss gradual degradation, so graceful degradation activates reactively rather than proactively.

Best practices

- [AGENTRELO8-BP01 Establish consistent configuration management practices](#)
- [AGENTRELO8-BP02 Implement agent tracing for telemetry throughout agent processing](#)
- [AGENTRELO8-BP03 Architect agent systems with resource isolation and contention mitigation](#)
- [AGENTRELO8-BP04 Track agent memory utilization metrics](#)

AGENTRELO8-BP01 Establish consistent configuration management practices

When different agent instances run with different configuration, the resulting reliability issues are hard to reproduce and harder to trust. Centralized configuration with versioning, validation, and automated distribution keeps every instance on the same current state and makes rollback a matter of changing a version pointer.

Desired outcome:

- You have centralized configuration with versioning and validation for every setting agents read dynamically.
- You have deployment strategies, gradual rollout for routine changes, immediate for emergencies, with automatic rollback on error.
- You detect configuration drift across the agent fleet and remediate it automatically.

Common anti-patterns:

- Hardcoding configuration in agent code, requiring redeployment to change values and reducing the risk of dynamic adjustment.
- Managing configuration without versioning, making it impossible to identify which change caused a regression or roll back cleanly.
- Applying configuration changes without validation, letting misconfigured values reach production.

Benefits of establishing this best practice:

- Agent behavior stays consistent across instances through centralized management.
- Configuration changes ship safely through validation and gradual rollout with rollback capability.
- Operational issues get resolved faster through dynamic adjustment without redeployment.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Centralized configuration is the unifying pattern. [Amazon Bedrock AgentCore Runtime](#)'s configuration capabilities manage agent settings centrally with versioning and validation. Runtime configuration that agents read dynamically includes model selection, tool availability, rate limits, feature flags, and operational thresholds. Use a managed configuration service with JSON Schema validators that enforce compliance before deployment. Validation at the configuration layer catches bad values before they become production incidents.

Deployment strategy keeps configuration changes safe. Gradual rollout handles the routine case. Propagate the new config to a small percentage of the fleet, watch for regressions, then expand. Automatic rollback on error reverses the change when something goes wrong. Immediate deployment handles the emergency case where the current configuration is actively breaking production and the cure can't wait for gradual rollout. Having both modes available, and knowing which one applies to each change, is what keeps the system responsive without being reckless.

Drift detection closes the loop. Configuration change detection in agent functions logs when versions change, enabling correlation of behavioral changes with specific deployments. For sensitive configuration values, use encrypted parameter storage with fine-grained access control. Monitor for configuration drift across the agent fleet through [Amazon Bedrock AgentCore Observability](#), alerting when instances are running with different configuration versions. Drift that persists is usually a sign that deployment rolled out partially or that a manual override was applied and forgotten.

Implementation steps

1. **Define configuration profiles per domain:** Build profiles for model selection, tool availability, rate limits, and feature flags. Apply JSON Schema validation to each profile.
2. **Configure deployment strategies:** Use gradual rollout for routine changes and immediate deployment for emergencies, with automatic rollback on error.

3. **Implement configuration change detection logging:** Log version changes so behavioral changes can be correlated with deployments.
4. **Use encrypted parameter storage for sensitive values:** Apply fine-grained access control on secrets.
5. **Monitor for configuration drift:** Use [Amazon Bedrock AgentCore Observability](#) to alert when instances run different configuration versions.

Resources

Related best practices:

- [AGENTREL06-BP05 Implement dynamic capability toggling](#)
- [AGENTREL08-BP02 Implement agent tracing for telemetry throughout agent processing](#)
- [AGENTREL08-BP03 Architect agent systems with resource isolation and contention mitigation](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Make agents a reality with Amazon Bedrock AgentCore: Now generally available](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Operationalizing agentic AI on AWS](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTREL08-BP02 Implement agent tracing for telemetry throughout agent processing

Request-boundary telemetry can't distinguish between normal variation and genuine degradation. Stage-level telemetry across the full processing lifecycle gives operators the signal they need to activate graceful degradation at the right time rather than too early or too late.

Desired outcome:

- You have stage-level telemetry across context retrieval, model inference, tool execution, and response generation.
- You capture LLM-specific data, token counts, inference latency, finish reason, on every model call.
- You have CloudWatch dashboards with stage-level widgets and composite alarms that drive automated degradation activation.

Common anti-patterns:

- Implementing telemetry only at the request boundary without instrumenting individual processing stages.
- Omitting LLM-specific telemetry, token counts, inference latency, output quality. That is essential for detecting model-related degradation.
- Treating telemetry as a post-deployment concern rather than designing it into the architecture from the start.

Benefits of establishing this best practice:

- Degradation detection is accurate because telemetry covers every processing stage.
- Degradation decisions are informed by which stage is actually under pressure.
- Incident response is faster through pre-built dashboards that expose the signals immediately.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock AgentCore Observability](#) captures agent-specific telemetry through OpenTelemetry-compatible instrumentation. This includes tool invocation traces, model interaction details, and memory access patterns. Stage-level instrumentation is what distinguishes this from generic request logging. A single latency metric at the request boundary can't tell you whether slowness came from retrieval, inference, tool execution, or response parsing.

LLM-specific telemetry deserves its own emphasis. Enable [Amazon Bedrock model invocation logging](#) to capture full request and response data for every LLM call, including token counts, latency, and finish reason. A spike in output token counts or a shift in finish-reason distribution

often precedes a visible quality regression by hours. Without the logs, those early signals are invisible.

The metric set needs structure. For each processing stage define a standard set. Context retrieval tracks latency and result count. Model inference tracks latency, token counts, and model ID. Tool execution tracks call count, latency, and error rate. Response generation tracks output validation pass rate. Build Amazon CloudWatch dashboards with stage-level widgets and a composite health summary so operators can see the full lifecycle at a glance. [Amazon CloudWatch Anomaly Detection](#) on the key metrics establishes baselines automatically, removing the need to hand-tune thresholds. Composite alarms across multiple stages detect degradation patterns that span boundaries and trigger automated degradation activation when the composite health signal drops.

Implementation steps

1. **Enable AgentCore Observability with stage-level telemetry:** Instrument context retrieval, inference, tool execution, and response generation. Use [Amazon Bedrock AgentCore Observability](#) for collection.
2. **Enable Amazon Bedrock model invocation logging:** Capture token counts, latency, and finish reason for every LLM call through [Amazon Bedrock model invocation logging](#).
3. **Build CloudWatch dashboards with stage-level widgets:** Include a composite health summary so operators see the full lifecycle at a glance.
4. **Configure CloudWatch Anomaly Detection on key metrics:** Use [Amazon CloudWatch Anomaly Detection](#) for automatic baseline modeling.
5. **Create composite alarms:** Combine multi-stage signals to trigger automated degradation activation.

Resources

Related best practices:

- [AGENTREL07-BP03 Implement distributed tracing to track system dependencies and facilitate recovery](#)
- [AGENTREL08-BP01 Establish consistent configuration management practices](#)
- [AGENTREL08-BP03 Architect agent systems with resource isolation and contention mitigation](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Build trustworthy AI agents with Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock model invocation logging](#)
- [Amazon CloudWatch Anomaly Detection](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)
- [Amazon CloudWatch](#)

AGENTREL08-BP03 Architect agent systems with resource isolation and contention mitigation

Shared resource pools let one noisy agent starve the rest. Priority tiers with dedicated resource allocations and contention detection keep user-facing agents responsive even when background workloads spike.

Desired outcome:

- You have separate runtime infrastructure for different agent priority tiers so high-priority agents have dedicated resources.
- You track token consumption for each agent and enforce per-agent access to shared model capacity.
- You detect contention early through composite signals and activate automated mitigation before failures occur.

Common anti-patterns:

- Sharing resource pools across every agent without isolation, letting high-volume agents consume resources needed by others.
- Skipping API quota management, so throttling affects every agent whenever any single agent exceeds quotas.
- Treating every agent as equally important, letting background workload spikes degrade user-facing agents.

Benefits of establishing this best practice:

- Performance stays predictable because resource isolation helps prevent cross-workload interference.
- Service quality for high-priority agents holds through priority-based resource allocation.
- Contention gets detected early through composite monitoring before it becomes a failure.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Isolation starts at the execution surface. Deploy separate [Amazon Bedrock AgentCore Runtime](#) instances for different agent priority tiers, so high-priority user-facing agents run on dedicated Runtime instances with their own resource allocations that background agents can't consume. This is the cleanest form of bulkheading for agent workloads, separate pools that physically can't interfere with each other, with no shared scheduler to introduce coupling.

Quota protection handles the shared-model case. Amazon Bedrock inference capacity is shared across the account. Track token consumption for each agent through [Amazon Bedrock AgentCore Observability](#) and Amazon CloudWatch alarms to catch individual agents approaching consumption thresholds. [Amazon Bedrock AgentCore Policy](#) Cedar policies control which agents can access which models. Combining policy with Amazon Bedrock service quotas and [Provisioned Throughput](#) helps prevent one agent from exhausting shared model capacity. For latency-sensitive agents that need predictable inference performance regardless of overall service demand, Provisioned Throughput gives you fixed model units and the predictable latency that goes with them.

With contention detection, you can act before the incident hits. Amazon CloudWatch composite alarms combine multiple resource utilization signals into a contention score. These signals include concurrency utilization, token consumption rates, and queue depths. When the score crosses the threshold, trigger automated mitigation. Use [Amazon Bedrock AgentCore Policy](#) to deny tool access for low-priority agents, or activate graceful degradation for non-critical capabilities. Monitor resource utilization across priority tiers through AgentCore Observability dashboards so emerging contention becomes visible before it causes user-visible failures.

Implementation steps

1. **Deploy separate AgentCore Runtime instances per priority tier:** Give high-priority user-facing agents dedicated [Amazon Bedrock AgentCore Runtime](#) resource allocations.

2. **Track per-agent token consumption and enforce access:** Use [Amazon Bedrock AgentCore Observability](#) and [Amazon Bedrock AgentCore Policy](#) to control model access per agent.
3. **Use Amazon Bedrock Provisioned Throughput for latency-sensitive agents:** Use [Provisioned Throughput](#) for predictable inference performance.
4. **Configure composite alarms on resource utilization signals:** Combine concurrency, token consumption, and queue depth signals through Amazon CloudWatch into a contention score.
5. **Implement automated contention mitigation:** Deny tool access for low-priority agents through AgentCore Policy when pressure is detected.

Resources

Related best practices:

- [AGENTREL01-BP05 Implement adaptive provisioning](#)
- [AGENTREL08-BP01 Establish consistent configuration management practices](#)
- [AGENTREL08-BP02 Implement agent tracing for telemetry throughout agent processing](#)
- [AGENTREL08-BP04 Track agent memory utilization metrics](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Policy](#)
- [Securely launch and scale your agents and tools on Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock Provisioned Throughput](#)
- [Amazon Bedrock AgentCore Observability](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)
- [Amazon CloudWatch](#)

AGENTREL08-BP04 Track agent memory utilization metrics

Memory exhaustion produces agents that look healthy but lack the context to reason well. Tracking utilization across short-term, long-term, and in-context tiers reveals the pressure before silent failures begin.

Desired outcome:

- You track token counts per context component and emit context-window utilization percentages.
- You have alarms when context window utilization exceeds 80%, triggering summarization or pruning workflows.
- You detect memory growth trends through metric math so gradual leaks surface before they cause failures.

Common anti-patterns:

- Monitoring only infrastructure-level memory metrics without tracking agent-specific patterns like context window utilization and session state growth.
- Operating without baselines for normal memory consumption, making anomalous growth undetectable.
- Skipping in-context memory utilization, the most direct indicator of context-related degradation.

Benefits of establishing this best practice:

- Memory pressure gets detected early through continual monitoring before exhaustion causes failures.
- Degradation decisions are informed by which memory tier is actually under pressure.
- Silent memory-related failures get prevented because in-context utilization is monitored proactively.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

In-context memory is where silent failures start. When the context window fills with retrieved context, conversation history, and tool results, the model has less room for new information and

its effective reasoning capacity drops. Tracking utilization by component (system prompt, retrieved context, conversation history, tool results) through [Amazon Bedrock AgentCore Observability](#) tells you which component is driving pressure. Alarms when utilization exceeds 80% of the model's context window trigger summarization or pruning workflows before the limit becomes a hard wall. For accurate token measurement with Anthropic models on Amazon Bedrock, use [Amazon Bedrock token counting](#). For other providers, approximate estimation works well enough for baseline purposes.

External memory stores are the other place pressure shows up. Monitor [Amazon Bedrock AgentCore Memory](#) access latency and error rates through AgentCore Observability, and watch infrastructure-level metrics through Amazon CloudWatch. Latency climbing on memory access is often the first signal that the store is under pressure, well before it actually fails.

Growth trend analysis catches the leaks infrastructure-level metrics miss. Use [Amazon CloudWatch Metric Math](#) to calculate growth rates over configurable windows and alert when the rate exceeds baseline. Steady incremental growth rarely trips a threshold alarm but often indicates a leak that will eventually exhaust memory. Build automated memory management responses for each tier. Apply context summarization for in-context pressure, session pruning for short-term memory pressure, and memory consolidation for long-term memory pressure.

Implementation steps

1. **Track in-context memory utilization per component:** Measure token counts for system prompt, retrieved context, conversation history, and tool results. Use [Amazon Bedrock AgentCore Observability](#) for collection.
2. **Configure alarms at 80% context window utilization:** Trigger summarization or pruning workflows before the limit becomes a hard wall.
3. **Monitor AgentCore Memory access latency and error rates:** Catch external store pressure early through [Amazon Bedrock AgentCore Observability](#).
4. **Implement memory growth trend analysis:** Use [Amazon CloudWatch Metric Math](#) to detect gradual leaks that don't trip threshold alarms.
5. **Build automated memory management responses:** Implement summarization, pruning, and consolidation responses per tier.

Resources

Related best practices:

- [AGENTREL03-BP04 Implement graceful degradation for memory and state operations](#)
- [AGENTREL08-BP01 Establish consistent configuration management practices](#)
- [AGENTREL08-BP02 Implement agent tracing for telemetry throughout agent processing](#)
- [AGENTREL08-BP03 Architect agent systems with resource isolation and contention mitigation](#)

Related documents:

- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock token counting](#)
- [Amazon CloudWatch Metric Math](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)
- [Amazon CloudWatch](#)

Performance efficiency

Agents that respond quickly, scale smoothly, and use resources efficiently earn user trust and drive adoption, slow or unresponsive agents get abandoned regardless of how accurate they are. Performance efficiency in agentic AI systems extends beyond traditional compute and network optimization to encompass the unique characteristics of autonomous, goal-directed agents that reason, plan, and act through large language models. Unlike conventional workloads where performance is primarily a function of infrastructure sizing, agentic AI performance is shaped by cognitive pipeline design, memory and context management, model selection strategies, multi-agent coordination patterns, and tool integration efficiency. This pillar provides best practices for optimizing every layer of the agentic stack, from the inference calls that power agent reasoning to the orchestration patterns that coordinate multi-agent workflows, so that agent systems deliver responsive and scalable performance aligned with business outcomes.

Capabilities

- [Strategic performance planning and measurement](#)
- [Core processing and reasoning pipeline optimization](#)
- [Memory, context, and RAG optimization](#)
- [Communication and protocol efficiency](#)
- [Workflow orchestration and multi-agent collaboration](#)
- [Tool integration and framework optimization](#)
- [Multi-tenancy and resource optimization](#)

Design principles

In addition to the lens-level design principles, the performance efficiency best practices in this lens are represented by at least one of the following principles:

- **Set targets per agent class, not per platform:** Streaming, task-oriented, and batch agents have different primary KPIs (time-to-first-token, completion time, throughput). Commit to the right ones for each class instead of a single platform-wide SLA.
- **Design and tune the reasoning pipeline against the latency budget:** End-to-end latency is the sum of inference, retrieval, tool calls, and handoffs. Profile where time actually goes, allocate per-phase budgets, and optimize the proven critical path rather than the assumed one.

- **Move work off the synchronous path:** Asynchronous messaging, streaming responses, parallel tool invocation, and event-driven coordination decouple user-perceived latency from total work performed.
- **Tier memory and retrieval to access patterns:** Hot context near the agent, warm in nearby caches, cold in durable stores. Retrieval cost (latency, compute, tokens) should match the recency and frequency of access, not the maximum.
- **Isolate tenants without serializing them:** Multitenant agent platforms need per-tenant throttling, quotas, and capacity reservations so heavy workloads cannot starve neighbors.

Strategic performance planning and measurement

Agentic AI systems introduce novel performance dimensions beyond traditional infrastructure metrics, including reasoning latency, token throughput, tool invocation efficiency, time-to-first-token, and overall task completion time. Strategic performance planning requires defining measurable success criteria with per-phase latency budgets, implementing thorough telemetry across the reasoning pipeline, systematically profiling overall latency to identify bottlenecks, and establishing performance testing practices that validate agent responsiveness under load. Without deliberate performance planning, organizations risk deploying agents that appear functional but fail to meet latency, throughput, or quality expectations at scale.

For agent maturity models and lifecycle governance, see [AGENTOPS03-BP01](#). For business metrics feedback loops, see [AGENTOPS02-BP04](#). For agent evaluation frameworks, see [AGENTOPS06-BP02](#).

AGENTPERF01: How do you plan strategically for agent performance and establish measurement practices?

Capability intent

- Performance success criteria are defined as measurable service level objectives per agent workload, covering latency (including time-to-first-token and end-to-end completion), throughput, quality, and efficiency dimensions that agent-specific reasoning, tool use, and multi-agent coordination introduce.

- End-to-end latency budgets are decomposed into per-phase allocations across context retrieval, inference, tool invocation, and inter-agent coordination, so that drift is attributable to the phase causing it rather than diagnosed during an incident.
- Distributed telemetry captures every agent execution as a session, trace, and span hierarchy aligned with OpenTelemetry generative AI semantic conventions, so reasoning-pipeline operations are individually observable and correlatable.
- Production traffic is profiled on a defined cadence to decompose end-to-end latency into per-phase contributions, with optimization effort ranked by contribution multiplied by addressable variance rather than by contribution alone.
- Success criteria, telemetry, and phase-level alerts feed CI/CD quality gates and regression alarms, so performance posture stays aligned with production behavior as prompts, tools, and models drift.

Maturity levels

These levels summarize what each stage of maturity looks like for strategic performance planning and measurement as a whole.

Level	Name	What it looks like
1	Initial	Performance targets are aspirational rather than measurable, with no per-workload KPIs and only infrastructure metrics in place. Telemetry is limited to runtime logs and Amazon CloudWatch duration metrics, and end-to-end latency is the only signal monitored . Performance regressions are discovered through user reports rather than observability, and criteria are revisited only after incidents.

Level	Name	What it looks like
2	Emerging	Latency and throughput targets are documented for flagship workloads, and basic traces flow through AWS X-Ray or CloudWatch. Agent-specific metrics (task completion, tokens per task) are emitted but are not currently tied to alerts or SLOs. Per-phase decomposition is one-time rather than encoded in criteria, and profiling happens reactively after the first incident.

Level	Name	What it looks like
3	Defined	Every production workload has structured service level objectives with service level indicators, thresholds, attainment goals, and intervals managed through Amazon CloudWatch Application Signals. OpenTelemetry-compatible telemetry through the AWS Distro for OpenTelemetry populates the CloudWatch generative AI observability surface, and per-phase latency budgets are allocated from the end-to-end target. Profiling runs against production traffic and per-phase percentile metrics are published.

Level	Name	What it looks like
4	Proactive	Phase-level CloudWatch anomaly detection and burn-rate alarms fire on drift before end-to-end SLOs are exceeded, with composite alarms correlating signals to reduce noise. SLOs are gated in CI/CD through automated load and evaluation tests, and Amazon Bedrock AgentCore Evaluations validates quality as part of the pipeline. Per-phase profiles are refreshed on a defined cadence and optimization targets are ranked by contribution and addressable variance.

Level	Name	What it looks like
5	Optimized	Success criteria, per-phase budgets, and thresholds are recalibrated continuously from production telemetry rather than through manual review cycles. Evaluation-set-driven quality gates and cost-per-task ceilings travel with every release, and profiling feeds the architecture roadmap. The organization contributes performance measurement patterns and reasoning-cost profiles back to its communities of practice and tunes targets from data rather than launch-time assumption.

Common issues to watch for

- Teams define success criteria in infrastructure terms and miss agent-specific dimensions such as reasoning latency, token efficiency, tool accuracy, and time-to-first-token, so the criteria can't detect the regressions that actually affect users.
- Single latency targets are applied across streaming and non-streaming agents, or across interactive and batch workloads, which either over-provisions some agents or sets unachievable goals for others.
- Telemetry is added after launch and leaves gaps in trace continuity across agent boundaries, tool invocations, and asynchronous operations, which makes performance investigations collapse into guesswork.
- End-to-end latency is measured without per-phase decomposition, so optimization effort lands on phases that contribute little to the tail rather than on the phase that actually bounds user-visible performance.

- Success criteria, telemetry, and profiling are treated as launch-time artifacts rather than as artifacts revisited on a cadence, which lets agent drift erode KPIs silently until users notice.

Best practices

- [AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads](#)
- [AGENTPERF01-BP02 Implement comprehensive performance telemetry](#)
- [AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets](#)

AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads

Agent workloads are harder to measure than traditional applications because a single user request fans out into multiple inference calls, tool invocations, and memory retrievals, each with its own latency, quality, and cost signature. Without explicit performance targets, teams optimize against a moving reference and can't tell when an agent is ready for production.

Desired outcome:

- You have documented performance success criteria for every agent workload, with specific, measurable targets that are reviewed as business requirements evolve.
- Your teams objectively assess whether an agent meets performance expectations before deployment and continually validate performance in production.
- You have performance criteria integrated into CI/CD pipelines as quality gates, helping prevent regressions from reaching production.

Common anti-patterns:

- Defining success criteria only around infrastructure metrics such as CPU utilization or memory consumption, without measuring agent-specific dimensions like reasoning latency, token efficiency, or task completion quality.
- Applying a single latency target across streaming and non-streaming agents, or across interactive and batch workloads, when time-to-first-token and end-to-end completion are primary KPIs for different agent classes.
- Establishing performance targets after deployment rather than during design, producing architectures that can't meet requirements without significant rework.

Benefits of establishing this best practice:

- Explicit success criteria establish concrete targets against which telemetry can be evaluated, making downstream performance work measurable rather than speculative.
- Performance-aligned criteria direct teams to optimize the reasoning pipeline for the metrics that matter, rather than pursuing generic optimizations that don't improve business outcomes.
- Quality gates tied to measurable targets convert success criteria into enforceable artifacts that block regressions from reaching production rather than detecting them after the fact.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Success criteria for an agent workload must be concrete enough to evaluate a build against and specific enough to drive architectural decisions. A performance-aligned criterion names the signal, the threshold, the percentile or attainment goal, the evaluation window, and the business outcome it helps protect. Without that specificity, teams can't decide whether to swap models, add caching, split a workflow, or reject a release, as there is no reference against which the change can be judged.

Agent workloads have a wider KPI surface than traditional applications because a single user request expands into multiple inference calls, retrieval queries, tool invocations, and, for multi-agent systems, inter-agent handoffs, each with its own latency, error mode, and cost. A complete set of success criteria spans four dimensions:

1. Latency (time-to-first-token for streaming agents, end-to-end completion time for task-oriented agents, and per-phase budgets across the reasoning pipeline)
2. Throughput (concurrent sessions, sustained requests per second, and queue depth under load)
3. Quality (task completion rate, tool selection and parameter accuracy, reasoning grounding, and response faithfulness)
4. Efficiency (tokens per task, cost per completion, and cache hit rate).

Layered [agent evaluation frameworks](#) decompose quality further into component-level signals that infrastructure-only monitoring can't see, like tool use, memory retrieval, multi-turn topic adherence, reasoning accuracy, responsibility, and safety.

The primary latency KPI differs by agent class. For streaming, conversational agents, users perceive responsiveness through time-to-first-token and inter-token latency, so those are the KPIs that gate releases. For task-oriented agents that return a single structured result, task completion time is primary and time-to-first-token is largely irrelevant. For batch or asynchronous workflows, throughput and cost-per-task dominate, and strict p99 latency matters less than predictable completion-within-SLA. A single latency target across these classes either over-provisions some agents or sets unachievable goals for others. [AWS Well-Architected Performance Efficiency guidance](#) reinforces that tail behavior matters more than the mean, p90 and p99 anchor latency targets because averages mask the slow responses that drive user-perceived poor experience.

A measurable target has more structure than a number. A service-level objective (SLO) combines a service-level indicator (the metric), a threshold, an attainment goal (the percentage of time or requests that must meet the threshold), an interval (calendar or rolling window), and an error budget (the allowable shortfall).

[Amazon CloudWatch Application Signals service level objectives](#) formalize this structure and add burn-rate alarms that fire when the budget is being consumed faster than expected, which matters for agent workloads whose latency distribution shifts subtly as prompts, tools, or models drift. SLOs should be decomposed into per-phase latency budgets, inference, retrieval, tool calls, inter-agent coordination, so that when a budget is exceeded, attribution to the offending phase is already encoded in the criteria rather than investigated after the fact.

Success criteria are not static. Agent behavior is shaped by prompts, memory, tools, and the models behind them, all of which drift, so criteria that are correct at launch erode as the system evolves. Integrating targets into CI/CD as quality gates, latency budgets checked against load-test results, task-completion rate checked against a curated evaluation set, cost-per-task checked against a ceiling, converts them from documents into enforceable artifacts that block regressions at the boundary.

[Operationalizing agentic AI on AWS](#) frames this lifecycle discipline as mandatory rather than optional, because continuous measurement is the only mechanism that keeps criteria aligned with the business outcomes they were designed to protect.

Implementation steps

1. **Capture business outcomes and user expectations with stakeholders:** Work with product, business, and user-experience owners to document the outcomes the agent must produce, the scenarios in which it operates, and the user expectations for responsiveness, quality, and cost. Use this intake to frame every downstream target so latency, throughput, quality, and efficiency

KPIs trace back to a stated business or user need. The [AWS Well-Architected Performance Efficiency process guidance](#) treats this stakeholder alignment as the first step in establishing credible KPIs.

2. **Classify each agent workload by interaction pattern:** Determine whether the agent is streaming or non-streaming, interactive or batch, synchronous or asynchronous, and single-agent or multi-agent. The classification dictates which latency KPI is primary, time-to-first-token for streaming conversational agents, end-to-end completion for task-oriented agents, throughput and cost-per-task for batch workflows, and whether multi-agent coordination metrics such as handoff latency and collaboration success belong in the criteria set.
3. **Define the KPI taxonomy spanning latency, throughput, quality, and efficiency:** For each workload, enumerate the specific signals to measure along the four dimensions, including agent-specific signals that infrastructure metrics can't cover. Structure quality signals in layers, final-response quality, task completion, tool use accuracy, memory and retrieval relevance, reasoning grounding, and responsibility and safety, following the [agent evaluation framework described by Amazon teams](#). Include efficiency signals such as tokens per task, cost per completion, and cache hit rate to connect performance to unit economics.
4. **Set quantitative targets with thresholds, percentiles, and attainment goals:** Attach a numeric target to every KPI, specifying the percentile (p50, p90, p99) for latency and throughput signals and the attainment goal (for example, 99.5 percent of requests) for quality and availability signals. Anchor latency targets to p90 and p99 rather than averages, because tail behavior drives user-perceived performance, the [AWS Well-Architected Performance Efficiency pillar](#) cautions that averages hide the slow responses that matter most.
5. **Allocate per-phase latency budgets within the end-to-end budget:** Decompose the end-to-end latency target into per-phase budgets, context retrieval, LLM inference, tool invocation, inter-agent coordination, output generation, so each phase has its own ceiling that sums to the overall target. Per-phase budgets make exceedances attributable at criteria-definition time and give engineering teams clear optimization targets when a phase drifts. Validate the decomposition against measured traces so the budgets reflect real behavior rather than assumption.
6. **Formalize targets as service level objectives with error budgets and burn-rate alerts:** For each customer-facing KPI, encode the target as an SLO with an SLI, threshold, attainment goal, interval, and period using [Amazon CloudWatch Application Signals service level objectives](#). Configure burn-rate alarms so the service signals when the error budget is being consumed faster than expected, giving operators time to respond before an SLO is exceeded. Group related

SLIs into composite SLOs where a single user-facing outcome depends on multiple operations meeting their individual targets.

7. **Operationalize quality and cost KPIs through GenAI-aware monitoring:** Publish token consumption, per-invocation latency percentiles, cost attribution, and agent-level quality metrics through [Amazon CloudWatch generative AI observability](#), which surfaces these signals natively for Amazon Bedrock model invocations and accepts structured traces from agents running on any runtime through the [AWS Distro for OpenTelemetry](#). Emit agent-specific quality signals, task completion rate, tool selection accuracy, reasoning grounding, as custom metrics so they can be thresholded, alarmed, and tied to SLOs the same way infrastructure signals are.
8. **Integrate targets as quality gates in the deployment pipeline:** Convert each success criterion into an automated check in CI/CD so releases that exceed a latency, quality, or cost target are blocked before they reach users. Run load tests against the latency and throughput targets and curated evaluation sets against the quality targets as part of the pipeline, following the lifecycle-management framing in [Operationalizing agentic AI on AWS](#). Gates turn written criteria into enforceable artifacts that help prevent regressions rather than detecting them in production.
9. **Revisit targets on a defined cadence as the workload evolves:** Schedule regular reviews of each success criterion against production telemetry, changes in user expectations, and shifts in the underlying models, prompts, or tools. Tighten targets that are consistently exceeded, relax targets that are blocking legitimate improvements, and retire signals that no longer map to a business outcome, agent behavior drifts, and criteria that were correct at launch need to be revalidated against current reality.

Resources

Related best practices:

- [AGENTPERF01-BP02 Implement comprehensive performance telemetry](#)
- [AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets](#)
- [AGENTOPS05-BP04 Define and track KPIs for agent workflows](#)
- [AGENTOPS06-BP02 Evaluate and track ongoing agent performance](#)
- [AGENCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)

Related documents:

- [AWS Well-Architected Framework: Performance Efficiency Pillar](#)
- [Amazon CloudWatch Application Signals: Service level objectives \(SLOs\)](#)
- [Amazon CloudWatch generative AI observability](#)
- [Operationalizing agentic AI on AWS](#)
- [Blog: Evaluating AI agents: Real-world lessons from building agentic systems at Amazon](#)

Related videos:

- [AWS re:Invent 2024 - Elevate application and generative AI observability \(COP326\)](#)

Related services:

- [Amazon CloudWatch](#)
- [Amazon CloudWatch Application Signals](#)
- [AWS Distro for OpenTelemetry \(ADOT\)](#)

AGENTPERF01-BP02 Implement comprehensive performance telemetry

A single agent request fans out into chains of inference calls, parallel tool invocations, memory lookups, and inter-agent communications. Infrastructure-only monitoring can't determine by itself which of those contributes to a slow or expensive response. Agent-aware telemetry decomposes execution into observable, attributable operations so that performance decisions are grounded in measured behavior.

Desired outcome:

- You have a complete distributed trace for every agent execution that decomposes total latency into its constituent operations.
- You have real-time dashboards that provide visibility into agent performance trends, with the resulting metrics feeding the alerting layer.
- You have historical telemetry data that supports capacity planning, model selection decisions, and architecture optimization through data-driven analysis.

Common anti-patterns:

- Relying only on infrastructure-level metrics such as function duration or API gateway latency without instrumenting the agent's reasoning pipeline, making it impossible to distinguish between slow inference and slow tool calls.
- Treating telemetry as an afterthought, producing gaps in trace continuity across agent boundaries, tool invocations, and asynchronous operations.
- Collecting telemetry data without establishing baselines, thresholds, or alerts, creating a data lake of metrics that nobody monitors or acts upon.

Benefits of establishing this best practice:

- Fine-grained performance data directs engineering effort toward the operations that materially contribute to end-to-end latency, helping prevent wasted cycles on components with negligible impact on user experience.
- Span-level attribution reduces mean-time-to-resolution for production incidents by pinpointing whether slow responses originate in inference, tool calls, retrieval, or inter-agent coordination.
- Historical telemetry data supports informed model selection, routing, and capacity planning by comparing latency, token, and cost profiles across models and architectures.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Comprehensive agent telemetry means capturing metrics, traces, and logs, as well as structuring them so every step of the reasoning pipeline is individually attributable. If any step of the agent's multiple and unique operations is opaque, performance investigations become less data-based and therefore less useful.

[OpenTelemetry \(OTel\)](#) is the portable substrate for this instrumentation, and its [generative AI semantic conventions](#) define standard attributes for LLM operations, model, input and output tokens, request parameters, and finish reasons so that spans remain comparable across frameworks, models, and runtimes. Using these conventions rather than framework-specific schemas keeps telemetry portable when an agent moves between Amazon Bedrock AgentCore Runtime, AWS Lambda, Amazon ECS, Amazon EKS, or self-hosted infrastructure, and helps prevent a rewrite every time the deployment target changes.

Agent telemetry has a natural three-tier hierarchy documented in [AgentCore Observability](#).

1. A *session* represents a complete user conversation
2. A *trace* represents one request-response cycle within the session
3. *Spans* represent discrete operations inside a trace, a reasoning iteration, an LLM call, a tool invocation, a memory lookup, a retrieval query, or an inter-agent handoff

Organizing telemetry against this hierarchy turns an opaque slow agent signal into an attributable execution tree where each span carries its own latency, token counts, error status, and contextual attributes. Session identifiers must flow through every span so a trace can be linked back to its conversation, and trace context must propagate across agent and tool boundaries using the [W3C Trace Context](#) standard so asynchronous and multi-service workflows remain a single connected graph.

Instrumentation approaches differ by runtime, but the resulting telemetry should look the same. For agents on [Amazon Bedrock AgentCore Runtime](#), the runtime auto-instruments agent code and emits OTel-compatible traces, runtime metrics (invocations, session count, latency, errors, CPU and memory usage), and structured logs to Amazon CloudWatch without additional work.

For agents on other runtimes, the [AWS Distro for OpenTelemetry \(ADOT\)](#) exports traces, metrics, and logs to CloudWatch using the same semantic conventions so both populations appear in a unified observability surface.

Framework-level instrumentation libraries such as OpenInference, OpenLLMetry, OpenLit, and Traceloop emit the reasoning-pipeline spans, reasoning iterations, prompt and response content, tool-selection decisions, that generic runtime instrumentation can't see. Select a framework based on the agent framework in use (for example, Strands Agents, LangChain, LangGraph, CrewAI, or LlamaIndex).

On the ingestion side, [Amazon CloudWatch Transaction Search](#) ingests 100 percent of spans as structured logs in the `aws/spans` log group and indexes a configurable percentage as trace summaries, supporting end-to-end trace search without forcing sampling at the span level. Enabling [CloudWatch Application Signals](#) on Amazon Bedrock API calls automatically populates OTel GenAI attributes, `gen_ai.system`, `gen_ai.request.model`, `gen_ai.usage.input_tokens`, `gen_ai.usage.output_tokens`, `gen_ai.response.finish_reasons`, so token, cost, and finish-reason analysis is available without hand-written spans.

Some signals may not be captured implicitly, such as task success and failure rate, cache hit rate, tool-selection accuracy, time-to-first-token, and cost per task. For these signals, emit them as

CloudWatch custom metrics from the agent or the OTEL collector, dimensioned uniformly (agent ID, workflow, environment, task type, model ID) so they can be correlated with the spans they originated from and consumed by dashboards, SLOs, and anomaly detection downstream.

Implementation steps

- 1. Define a standardized telemetry schema:** Document the span types the agent will emit (reasoning iteration, LLM inference, tool invocation, memory operation, retrieval, inter-agent handoff) and the required attributes for each, aligning LLM spans with the [OpenTelemetry generative AI semantic conventions](#) and agent spans with the [GenAI agent span conventions](#). Specify a consistent set of metric dimensions, agent ID, workflow, environment, task type, model ID, that every custom metric and span must carry so signals remain correlatable across the stack.
- 2. Instrument the reasoning pipeline across every execution layer:** Wrap reasoning iterations, LLM inference, tool invocations, memory operations, retrieval queries, and inter-agent handoffs as OpenTelemetry spans. For agents on Amazon Bedrock AgentCore Runtime, the runtime auto-instruments agent code when [AgentCore Observability](#) is enabled. For agents on other runtimes, use the [AWS Distro for OpenTelemetry \(ADOT\)](#) together with a framework-specific instrumentation library (OpenInference, OpenLLMetry, OpenLit, or Traceloop) so reasoning-pipeline spans are captured rather than only request-level spans.
- 3. Propagate trace and session context across every boundary:** Carry [W3C Trace Context](#) headers through every outbound call the agent makes, tool invocations, retrieval queries, inter-agent handoffs, asynchronous queue-backed work, so a single user request produces a single connected trace rather than disconnected fragments. Propagate the session identifier through OpenTelemetry baggage so every span in a conversation can be linked back to its session for conversation-level analysis.
- 4. Enable CloudWatch ingestion for spans and Amazon Bedrock API attributes:** Complete the one-time setup for [CloudWatch Transaction Search](#) so 100 percent of spans are ingested as structured logs and a configurable percentage are indexed as trace summaries. Enable [CloudWatch Application Signals with GenAI attribute support](#) to auto-populate OTEL GenAI attributes on Amazon Bedrock API calls so token, model, and finish-reason data is captured without custom instrumentation.
- 5. Emit custom metrics for signals not captured natively:** Publish task success and failure rate, cache hit rate, tool-selection accuracy, time-to-first-token, and cost per task as [CloudWatch custom metrics](#) from the agent runtime or the OTEL collector, using the standard dimension set defined in step 1. Without these, observability tooling can display infrastructure and inference signals but can't answer product-level questions about agent quality or unit economics.

6. **Build the observability surface:** Use the [Amazon CloudWatch generative AI observability dashboard](#) to provide session, trace, and agent views for incident triage and trend analysis, and publish composed dashboards for per-workflow, per-model, and per-tenant slices using the standard dimension set.

Resources

Related best practices:

- [AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads](#)
- [AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets](#)
- [AGENTOPS05-BP01 Establish end-to-end tracing and telemetry for agent operations](#)
- [AGENTOPS05-BP05 Create workflow-specific dashboards for operational health](#)
- [AGENTCOST05-BP02 Implement distributed cost tracing for multi-agent workflows](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Blog: Build trustworthy AI agents with Amazon Bedrock AgentCore Observability](#)
- [Amazon CloudWatch generative AI observability](#)
- [CloudWatch Application Signals: Troubleshoot generative AI applications with OpenTelemetry GenAI attributes](#)
- [AWS Distro for OpenTelemetry \(ADOT\)](#)
- [Blog: Observing agentic AI workloads using Amazon CloudWatch](#)

Related videos:

- [AWS re:Invent 2024 - Observability for Reliable Agentic AI with Strands & OpenTelemetry \(NTA406\)](#)

Related examples:

- [GitHub: Amazon Bedrock AgentCore samples, Observability tutorials](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [Amazon CloudWatch Application Signals](#)
- [AWS Distro for OpenTelemetry \(ADOT\)](#)

AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets

The dominant contributor to agent latency varies by task type: a simple question-and-answer request can be inference-bound, a retrieval-heavy request can be retrieval-bound, and a multi-agent workflow can be coordination-bound. Without decomposing total latency into per-phase contributions, teams optimize assumed bottlenecks rather than measured ones, and engineering effort lands on work that doesn't move performance for users.

Desired outcome:

- You have a latency profile for every agent workload that decomposes latency into per-phase contributions, with the dominant phase identified and targeted for optimization.
- Your teams diagnose performance issues by examining the phase breakdown rather than guessing at which component is slow.
- You have per-phase regression alerts that fire on phase-level drift before the end-to-end service level objective is exceeded.

Common anti-patterns:

- Measuring only total latency without decomposing it into phases, making it impossible to tell whether slowness is caused by inference, retrieval, tool calls, or coordination overhead.
- Optimizing inference latency (model selection, prompt compression) when the actual bottleneck is retrieval or tool call latency, wasting engineering effort on a phase that contributes a small fraction of total time.

- Profiling under synthetic test conditions without validating against production traffic patterns, missing bottlenecks that only appear under concurrent load.

Benefits of establishing this best practice:

- Phase-level visibility directs engineering effort at the actual bottleneck rather than assumed bottlenecks.
- Per-phase trend monitoring pinpoints which phase degraded, making latency regressions faster to diagnose.
- Before-and-after phase profiles validate that an optimization actually shrank the targeted phase without regressing others.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Profiling an agent is the operation of aggregating span-based telemetry the reasoning pipeline already emits into contributions per phase, and then asking which phase dominates the budget. The decomposition is a grouping over existing trace data, provided every span maps cleanly to a phase in a shared taxonomy. When that mapping is ambiguous, a single span that covers both retrieval and output formatting, or a reasoning iteration that hides an inline tool call, the profile becomes untrustworthy and optimization decisions drift. The pre-work is in defining the phases once, aligning span types to them, and making the mapping visible to every team contributing to the agent.

The dominant phase varies by workload and shifts over time. For example:

- A conversational question-and-answer agent is usually inference-bound
- A retrieval-heavy agent that reads across several knowledge bases can be retrieval-bound
- A multi-agent workflow that serially hands off context between supervisor and workers is typically coordination-bound
- An agent that invokes external APIs can be tool-bound when a downstream service degrades

Optimizing the wrong phase produces no measurable user-facing improvement. For example, a 30 percent inference speedup is invisible when inference accounts for 10 percent of the budget and retrieval accounts for 60. Phase-level attribution helps prevent that mistargeting, and it remains

necessary even after a workload has been tuned because the dominant phase at launch rarely stays dominant after prompts, tools, and models evolve.

Compute profiles at the distribution level, not the mean. Averages hide the tail, and tail latency is what users experience when an agent feels slow. Compute p50, p90, and p99 for each phase separately, and for total latency. The dominant phase at the median is often not the dominant phase at p99, because the slow tail typically concentrates in one or two phases, inference during model throttling, retrieval during index rebuilds or cold caches, tool calls during downstream-service incidents. A profile that reports means only by phase can point a team at the wrong target, because the phase that hurts users at the tail is usually smaller than the phase that dominates the average.

Run profiling against production traffic patterns for credible results. Synthetic load tests can exercise the request path, but they rarely reproduce the prompt distributions, tool-selection behaviors, and concurrency patterns real users generate. This means that bottlenecks that only appear under contention stay invisible, like thread-pool starvation during fan-out, cache-miss bursts after warmup expires, or queueing in shared inference endpoints.

With [Amazon CloudWatch Transaction Search](#) ingesting 100 percent of spans to the aws/spans log group, you can reconstruct a production profile from real traffic using [CloudWatch Logs Insights](#) queries over span durations without standing up dedicated profiling infrastructure. Agents on [Amazon Bedrock AgentCore Runtime](#) produce compatible span data automatically when AgentCore Observability is enabled, and agents on Amazon ECS, Amazon EKS, AWS Lambda, or self-hosted infrastructure produce it through the [AWS Distro for OpenTelemetry](#) collector, so both populations share a single surface.

Rank optimization targets by contribution multiplied by addressable variance, not by contribution alone. A phase that contributes 40 percent of the budget but is already near its theoretical floor offers less headroom than a phase that contributes 25 percent but has high variance driven by implementation choices, sequential retrieval that could be parallelized, or tool calls that could be cached.

Profiling also makes it possible to detect regression by phase. Once a baseline distribution exists for each phase, [Amazon CloudWatch anomaly detection](#) alarms on a phase drifting outside its expected band before the change shows up in the SLO, so regressions are attributed to the offending phase at alert time rather than during an incident retrospective.

Implementation steps

1. **Define the phase taxonomy for the workload:** Enumerate the phases that make up end-to-end execution, input processing, context and memory retrieval, LLM inference, tool invocation, output generation or streaming, and inter-agent coordination, and map every span type emitted by the agent to exactly one phase, aligning span attributes with the [OpenTelemetry generative AI semantic conventions](#). Document the taxonomy so every team contributing spans groups them the same way, and add workload-specific phases such as guardrail evaluation or structured-output post-processing when they sit outside the common set.
2. **Aggregate spans into per-phase duration metrics:** Derive per-phase durations from the span log group using [CloudWatch Transaction Search](#) and [CloudWatch Logs Insights](#) queries that sum span durations grouped by phase and trace, then emit the result as [CloudWatch custom metrics](#) with a phase dimension. Publishing the profile as metrics, rather than only as ad-hoc log queries, lets the same signal flow into dashboards, alarms, and anomaly detection alongside native runtime metrics.
3. **Compute percentile-level profiles for every phase:** Calculate p50, p90, and p99 per phase and for end-to-end latency over a rolling window that matches the service level objective interval, and display the percentiles side by side. The dominant phase at the median is often not the dominant phase at p99, and profiling against only the mean hides the tail where user-perceived slowness concentrates.
4. **Visualize the contribution of each phase to end-to-end latency:** Build a stacked per-phase contribution view on a [customized CloudWatch dashboard](#) using the per-phase metrics emitted in the previous step, and use [dashboard variables](#) to pivot the same view across the standard dimension set, agent ID, workflow, environment, task type, model ID, so dominant-phase analysis runs per slice rather than against a fleet average that can obscure tenant-specific or workflow-specific bottlenecks. Pair the custom view with the pre-built [Amazon CloudWatch generative AI observability](#) views for session, trace, and span drill-down when a specific slow request needs investigation.
5. **Profile under production traffic rather than only synthetic tests:** Run the profile against real production trace data so concurrency effects, prompt-mix variance, and downstream contention appear in the distribution. Use synthetic load from tools such as the [Distributed Load Testing on AWS](#) solution to stress-test specific hypotheses, for example, to confirm that a proposed parallel-retrieval change removes an observed p99 tail, and anchor the phase-contribution view itself to production traffic so decisions reflect how users actually exercise the agent.
6. **Rank optimization targets by contribution and addressable variance:** For each phase, estimate the contribution to the end-to-end budget and the fraction of that contribution that

is addressable by engineering effort. Prioritize phases where both are high, a large contribution with room to shrink, over phases that are large but already near their theoretical floor, so engineering time moves user-visible latency rather than a metric that doesn't affect the tail.

7. **Set per-phase baselines and alarm on drift:** Establish a steady-state baseline distribution for each phase, then apply [CloudWatch anomaly detection](#) to the per-phase percentile metrics so deviations fire with the offending phase already attributed. Pair the phase-level alarms with the end-to-end service level objective so a phase drifting inside its per-phase budget triggers investigation before the end-to-end budget is exceeded.
8. **Measure before and after every optimization:** Capture the profile for each phase before an optimization is rolled out and compare it against the same profile under production traffic after the change lands. Validate that the targeted phase actually shrank, check that no other phase regressed to absorb the budget, and retain the comparison in the change record so future regressions can be diagnosed against a known-good profile.
9. **Revisit the profile as the workload evolves:** Agent behavior drifts as prompts, tools, memory, and models change, and the dominant phase at launch rarely stays dominant six months later. Refresh the phase-based profile after every significant prompt or model change and at least quarterly, then re-rank optimization targets against the current profile so engineering effort continues to land on the phase that actually limits user-visible performance.

Resources

Related best practices:

- [AGENTPERF01-BP01 Define performance-aligned success criteria for agent workloads](#)
- [AGENTPERF01-BP02 Implement comprehensive performance telemetry](#)
- [AGENTPERF02-BP01 Design efficient reasoning pipelines](#)
- [AGENTPERF02-BP03 Optimize agent execution paths for reduced latency](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Blog: Build trustworthy AI agents with Amazon Bedrock AgentCore Observability](#)
- [Amazon CloudWatch generative AI observability](#)
- [Observability and monitoring, Building serverless architectures for agentic AI](#)

Related videos:

- [AWS re:Invent 2024 - Elevate application and generative AI observability \(COP326\)](#)

Related examples:

- [GitHub: Amazon Bedrock AgentCore samples, Observability tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Core processing and reasoning pipeline optimization

A well-optimized reasoning pipeline is what makes agents feel fast, responsive, and worth using. It is the difference between an agent that users rely on and one they abandon. The reasoning pipeline (encompassing perception, reasoning, planning, decision-making, and action execution) is the performance-critical path of every agentic AI system. Each iteration of the agent loop involves an LLM inference call, which is typically the most latency-intensive and resource-consuming operation in the stack. Optimizing core processing requires designing efficient reasoning pipelines that minimize unnecessary iterations, selecting models appropriate to task complexity, reducing execution path latency through architectural decisions, and optimizing streaming delivery to minimize perceived latency for user-facing interactions. Poor pipeline design produces agents that are slow, expensive, and unresponsive, regardless of how well the underlying infrastructure is provisioned.

For the actor model as a foundational execution pattern, see [AGENTRELO1-BP03](#).

AGENTPERF02: How do you optimize core agent processing and reasoning pipelines?

Capability intent

- Reasoning pipelines are bounded by iteration caps and confidence-based early termination so simple tasks resolve in one or two iterations while complex tasks receive the iterations they need.

- Each task class is routed to the smallest model that meets its quality bar, with cascading fallback to a more capable model when the assigned model produces low-confidence outputs.
- Independent operations execute concurrently, connections and runtimes are warm across invocations, and repeated lookups within a single request are deduplicated through request-scoped caches.
- User-facing agents stream tokens with sub-second time-to-first-token, pre-inference work is compressed to preserve the TTFT budget, and tool invocations mid-stream surface progress to the user rather than unexplained pauses.
- Retry strategies and graceful degradation paths are bounded by explicit latency budgets so failure recovery stays inside the end-to-end service level objective rather than eroding it silently.

Maturity levels

These levels summarize what each stage of maturity looks like for core processing and reasoning pipeline optimization as a whole.

Level	Name	What it looks like
1	Initial	Agents run without iteration limits or confidence-based termination, and a single large model serves every task regardless of complexity. Independent operations run sequentially, connections are re-established per invocation, and user-facing agents wait for the complete response before any output reaches the user. Failures propagate without explicit retry budgets.
2	Emerging	Iteration caps and basic retry limits are in place for flagship agents, and task classification is documented but not consistently routed.

Level	Name	What it looks like
		Streaming is enabled for some user-facing interactions but the pre-inference path isn't optimized for time-to-first-token. Model selection happens at design time rather than benchmarked against the workload's task distribution.
3	Defined	Task classes are mapped to model tiers benchmarked on the workload's own distribution through Amazon Bedrock AgentCore Evaluations , with cascading fallback configured for low-confidence outputs. Independent operations execute concurrently inside framework primitives such as the Strands Agents , and Amazon Bedrock streaming APIs deliver output token-by-token. Retry strategies are bounded by explicit latency and token budgets.

Level	Name	What it looks like
4	Proactive	Model assignments and routing rules are externalized as runtime configuration in AWS AppConfig and promoted through progressive rollouts gated by CloudWatch alarms. Connection pooling through Amazon RDS Proxy and warm runtimes through Amazon Bedrock AgentCore Runtime sessions or AWS Lambda provisioned concurrency remove cold starts on the critical path. Tool invocations during streaming surface structured progress events to the client.

Level	Name	What it looks like
5	Optimized	Pipeline shape, caps, model routing, and fallback rules are recalibrated continuously from production telemetry , and Amazon Bedrock latency-optimized inference is evaluated for each eligible model. Voice and real-time workloads run on Amazon Bedrock AgentCore Runtime bi-directional streaming through Amazon Nova Sonic. Reasoning iteration counts, TTFT, and fallback-escalation rates sit alongside latency and cost on every dashboard , and the organization contributes pipeline patterns back to its communities of practice.

Common issues to watch for

- Agents reason without iteration caps or early termination signals, producing runaway loops that consume tokens and time without improving output quality.
- A single large model serves every task regardless of complexity, paying the latency and cost premium of a heavyweight model for work a smaller one resolves as well.
- Independent operations execute sequentially and connections are re-established per invocation, so end-to-end latency becomes the sum of every operation duration plus repeated setup overhead.
- User-facing agents wait for the complete response before any output reaches the user, so perceived latency equals total processing time rather than the shorter time-to-first-token streaming would deliver.

- Tool invocations mid-stream pause the output without user-visible progress, creating perceived stalls where users see partial output followed by silence for several seconds.

Best practices

- [AGENTPERF02-BP01 Design efficient reasoning pipelines](#)
- [AGENTPERF02-BP02 Implement task-appropriate model selection strategies](#)
- [AGENTPERF02-BP03 Optimize agent execution paths for reduced latency](#)
- [AGENTPERF02-BP04 Optimize streaming responses and time-to-first-token for agent interactions](#)

AGENTPERF02-BP01 Design efficient reasoning pipelines

Each iteration of the perceive-reason-act loop typically involves an LLM inference call, so the number of iterations an agent takes multiplies both latency and cost. An efficient pipeline reaches accurate decisions in the fewest iterations the task requires, uses bounded iteration limits and confidence-based early termination to help prevent runaway loops, and handles retries within explicit performance budgets rather than silently eroding them.

Desired outcome:

- You have per-task iteration limits and confidence-based early termination configured, so reasoning loops can't run without bounds.
- You have pipeline shapes that scale reasoning depth to task complexity, with simple tasks resolving in one or two iterations and complex tasks receiving the iterations they need.
- You have retry strategies bounded by explicit latency budgets, with semantic re-prompting or graceful degradation engaged before the end-to-end SLO is exceeded.
- You have average reasoning iterations per task tracked as a first-class KPI, visible alongside latency and token metrics.

Common anti-patterns:

- Allowing agents to reason indefinitely without iteration limits or early termination conditions, producing runaway loops that consume tokens and time without improving output quality.

- Designing reasoning pipelines that always execute the same sequence of steps regardless of task complexity, applying heavyweight reasoning to simple tasks that could be resolved with a single inference call.
- Designing retry strategies without performance budgets, so exponential backoff retries accumulate latency that exceeds the end-to-end SLO when semantic re-prompting or graceful degradation would preserve performance targets.
- Retrying a failed LLM call with the identical prompt and model, rather than re-prompting semantically, rephrasing the instruction, simplifying the task, or falling back to a more capable model, to increase the chance that the retry succeeds inside the remaining latency budget.

Benefits of establishing this best practice:

- Efficient pipeline design reduces the number of LLM inference calls per task, which is one of the highest-impact optimizations for agent latency and cost.
- Adaptive pipeline shape makes simple tasks resolve quickly while complex tasks receive the iterations they need.
- Explicit retry budgets and semantic re-prompting keep tail latency within SLO even when a tool or model call fails on the first attempt.
- Iteration caps produce tighter latency distributions that simplify capacity planning, auto scaling thresholds, and cost forecasting.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Total latency and token spend scale with the number of iterations the pipeline takes to reach an accepted output, so pipeline design, how the loop is structured, when it terminates, and how failures are handled has a multiplicative effect on both latency and cost. A well-designed pipeline reaches an accepted output in the fewest iterations the task actually requires.

The right pipeline shape depends on task complexity and predictability. [Basic reasoning agents](#) handle single-turn classification, extraction, or summarization in one inference call and should not be wrapped in iterative reasoning loops. [ReAct-style loops](#), in which the agent interleaves reasoning, tool calls, and observation, fit open-ended tasks where the next step can't be predicted at design time.

Plan-then-execute shapes such as ReWOO and plan-and-solve hybrids separate planning from execution and bound iteration by plan length rather than by the model's willingness to keep looping, and reflect-and-revise shapes such as Reflexion introduce explicit critique cycles with hard caps on revision passes. Both patterns are described in [Customize agent workflows with advanced orchestration techniques using Strands Agents](#).

Iteration caps and early termination are the two controls that keep a loop from consuming resources without producing value, and they serve complementary roles. A per-task iteration cap is an upper bound that helps prevent pathological runaway (a model that keeps calling tools without converging), and it must be set per task class because a ceiling appropriate for multi-step research is wasteful for a simple lookup.

Early termination ends the loop before the cap when additional iterations would not improve the output. For example, when a critique step returns a structured "no further revision needed" signal, when the agent's own confidence assessment exceeds a threshold, or when a deterministic validator (schema check, grounding check, policy check) confirms the output is acceptable.

Together, caps help prevent the worst case while termination removes the common case of paying for unnecessary iterations.

Failure recovery is part of the latency budget, not an exception to it. When a tool call or model inference fails, naive exponential backoff can consume more time than the end-to-end target allows. Every retry strategy should be bounded by an explicit performance budget that specifies how much latency and how many tokens retries can consume before the pipeline falls back to a degraded response.

Identical retries against the same prompt and model frequently fail for the same reasons:

- Semantic re-prompting (rephrasing the instruction, simplifying the task, or tightening the output contract)
- Model escalation (routing the retry to a more capable model)
- Tool substitution (using an alternative data source)

Each of these failures change a variable in the failure mode and increase the chance that the retry succeeds within the remaining budget. To preserve user trust, implement strategies like graceful degradation, returning a partial answer with marked gaps, a lower-confidence answer with explicit uncertainty, or a clear "can't complete" response before the latency target is exceeded.

Average reasoning iterations per task belongs alongside latency, tokens, and cost as a first-class performance KPI. It is the earliest signal that pipeline shape, prompt quality, or upstream tool reliability has drifted, because a rising iteration count typically precedes a latency or cost regression. Each extra iteration compounds with the others downstream before the user-facing metric shifts enough to trigger an alarm.

Tracked by task class and by pipeline shape, iteration count also reveals misrouted tasks like simple requests running through heavyweight loops or complex tasks capped before they converge. Both of these patterns are invisible to metrics that only measure the end result.

Implementation steps

- 1. Classify each agent task by reasoning complexity:** Group tasks by the reasoning depth they require, single-step extraction or classification, multi-step reasoning over known steps, and open-ended investigation where the path isn't knowable in advance. Use this classification as the input to pipeline-shape selection and iteration budgets, because applying the same shape and budget to every class either wastes iterations on simple work or under-reasons on complex work. Document the classification alongside the workload's success criteria so routing decisions can be audited and revisited as task distributions change.
- 2. Select a pipeline shape that matches each task class:** Map each class to a reasoning pattern documented in [Agentic AI patterns and workflows on AWS](#), a [basic reasoning pattern](#) for single-step tasks, a [ReAct loop](#) for open-ended reasoning where tool use drives the next step, and a plan-then-execute or reflect-and-revise shape for tasks that benefit from an explicit planner or critique stage as described in [Customize agent workflows with advanced orchestration techniques using Strands Agents](#). Avoid wrapping single-step tasks in iterative loops, which inflates cost and latency with no accuracy gain.
- 3. Set per-task iteration caps sized to the complexity class:** Configure a hard maximum on reasoning iterations for each task class using the iteration-control primitive exposed by the agent framework in use, and size the cap to a value the workload will hit only on pathological cases. Caps are a floor on the worst case, tasks that converge early still terminate early through the confidence signals configured next, so tune caps to the most complex variant of each class rather than to the typical case.
- 4. Define early-termination conditions so loops stop when iterations stop adding value:** Specify structured signals that end the loop before the cap, a critique step returning a boolean "revision needed" flag, a confidence score exceeding a defined threshold, or a deterministic validator (schema, grounding, or policy check) confirming the output is acceptable. Treat these signals as

data the pipeline produces and logs, not as implicit model behavior, so termination decisions are observable and auditable rather than hidden inside a chain of thought.

5. **Establish a retry budget bounded by the end-to-end latency target:** Allocate an explicit portion of the end-to-end latency target to retry handling and enforce it at the pipeline level so accumulated retries can't silently exceed the target. Decompose the budget into latency and token components, because a retry that stays inside the latency budget but triples token consumption still degrades unit economics. Align the budget with the service level objective the workload is graded on so retry behavior is measured against the same target as everything else in the pipeline.
6. **Replace identical retries with semantic re-prompting, model escalation, or tool substitution:** When a call fails, retry with a rephrased instruction, a simpler task decomposition, a more capable model, or an alternative tool, each changes a variable in the failure mode instead of repeating the same failing call. Select the substitution based on the failure signal: a timeout suggests model escalation or tool substitution, a parsing failure suggests re-prompting with a stricter output contract, and a grounding failure suggests retrieval expansion.
7. **Configure graceful degradation paths that return before the target is exceeded:** Define the fallback response for each task class, a partial answer with marked gaps, a lower-confidence answer with explicit uncertainty, or a clear "unable to complete" response, and invoke the fallback when the retry budget is exhausted or the latency budget is within a configured safety margin of being exceeded. Predictable tail behavior and a clear failure response preserve user trust better than maximizing the chance of an eventual success on every request.
8. **Emit reasoning iterations, retries, and terminations as first-class telemetry:** Publish iteration count, termination cause (early termination, cap hit, retry-budget exhausted, graceful degradation), and retry count for every invocation as metrics that can be thresholded and alarmed alongside latency and token metrics using a capability such as [Amazon CloudWatch generative AI observability](#) or an equivalent pipeline on the agent's runtime. Put average iterations per task on the same dashboards as latency percentiles and cost per completion, since it is the earliest indicator that pipeline shape, prompt quality, or tool reliability has drifted.
9. **Review pipeline shape, caps, and budgets against production telemetry on a defined cadence:** Schedule regular reviews of iteration distributions, early-termination rates, retry-budget consumption, and degradation frequency so pipeline parameters track actual behavior rather than launch-time assumption. Tighten caps that are consistently under-utilized, relax caps that are being hit on legitimate complex tasks, and re-classify tasks whose iteration distribution reveals they belong to a different complexity class than originally assigned.

Resources

Related best practices:

- [AGENTPERF02-BP02 Implement task-appropriate model selection strategies](#)
- [AGENTPERF02-BP03 Optimize agent execution paths for reduced latency](#)
- [AGENTPERF03-BP02 Optimize context window utilization and prompt management](#)

Related documents:

- [Blog: Customize agent workflows with advanced orchestration techniques using Strands Agents](#)
- [Foundations of agentic AI on AWS](#)
- [Agentic AI patterns and workflows on AWS](#)
- [Agentic AI - Generative AI Lens \(Well-Architected Framework\)](#)
- [Amazon CloudWatch generative AI observability](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon CloudWatch](#)

AGENTPERF02-BP02 Implement task-appropriate model selection strategies

Agent tasks vary widely in the reasoning they require, but default routing often sends every task to the same large model, paying the latency cost of a heavyweight model even for work a smaller one resolves as well. Matching model capability to task demand is one of the highest-use performance decisions in an agent system, because inference latency and throughput scale directly with model size. Done systematically, model selection reduces per-task latency without sacrificing quality on complex reasoning.

Desired outcome:

- You have agent tasks classified by reasoning complexity, with each class mapped to the smallest model that meets its quality bar.
- You have routing logic that directs each request to its assigned model at runtime, with a cascading fallback to a more capable model when the assigned model produces low-confidence or failed outputs.
- You have model assignments validated against benchmarked quality and latency on the workload's own task distribution rather than on generic leaderboard rankings.
- You have model selection treated as a runtime-configurable parameter so new model releases can be evaluated and rolled out without redeploying the agent.
- You have inference latency and task quality tracked per task class, so routing decisions are continually validated against both.

Common anti-patterns:

- Using a single large model for every agent task regardless of complexity, paying the latency and cost premium of a heavyweight model on work a smaller one resolves as well.
- Using a small or general-purpose model for tasks that require deeper reasoning, producing low-quality outputs that trigger retries, extra reasoning iterations, or manual escalation and eroding the latency savings the small model was meant to provide.
- Selecting models from general benchmark rankings rather than from benchmarks run on the workload's own task distribution, producing choices that are optimal for the leaderboard but suboptimal for actual traffic.
- Treating model choice as a one-time architectural decision baked into application code, so evaluating or rolling out a newly released model requires a code change and redeploy rather than a configuration update.
- Operating without a fallback path when the assigned model returns low-confidence or failing outputs, forcing the pipeline to return a poor answer or fail entirely rather than escalating that request to a more capable model.

Benefits of establishing this best practice:

- Routing lightweight work to smaller, faster models avoids paying the inference time of a large model for tasks that don't require it.
- Explicit routing to capable models and cascading fallback when a smaller model underperforms protect user-facing accuracy on complex tasks.

- Benchmarks against the workload's actual task distribution continually improve assignments as new evaluation data accumulates.
- Runtime-configurable selection lets capability and latency improvements from new models reach production without redeploy cycles.

Level of risk exposed if this best practice is not established: High

Implementation guidance

The reasoning pipeline's dominant cost component is the inference call, and inference latency and throughput scale directly with model size. Typical agent workloads have heterogeneous task mixes and simple classification alongside multi-step reasoning, so applying the largest model uniformly forces the blended latency and cost to track the most capable option even though most requests don't need it. Systematic model selection shifts this by assigning each task to the smallest model that still meets its quality bar.

Two approaches fit together:

1. Explicit task classification, where the agent assigns each task to a class and each class to a model or model tier, gives fine-grained control and works across providers and model families.
2. Managed routing such as [Amazon Bedrock intelligent prompt routing](#) predicts response quality per request within a model family and routes to the smallest model predicted to meet the quality bar, which hands off the per-request pick when the decision is purely capability-compared-to-cost inside one family.

Many workloads combine both. The agent picks the tier by task type while the router selects the specific model within that tier. Purpose-built services such as [Amazon Bedrock Data Automation](#) for intelligent document processing are also worth considering as a model for document-heavy tasks, where a dedicated pipeline is typically faster and more accurate than routing documents through a general-purpose vision model.

Routing decisions are only as good as the benchmarks behind them. Public leaderboards rank models on averages across heterogeneous benchmarks whose task distributions can bear little resemblance to a specific workload, so a model that leads a general benchmark can underperform on the traffic a particular agent actually serves.

[Amazon Bedrock AgentCore Evaluations](#) provides built-in evaluators, LLM-as-a-Judge, ground-truth correctness, and trajectory matching that make it practical to benchmark candidate models

against an evaluation set representative of actual traffic. Without workload-specific evaluation, routing choices are effectively guesses.

Selection is the starting point of a request, not the end of the decision. Cascading fallback preserves quality without forcing the whole task class to the larger model. Fallback differs from retry: retries repeat the same call hoping for a better draw, while fallback changes a variable by switching models. To promote new models through progressive rollouts with automated rollback, hold model identifiers, tier mappings, and fallback rules as runtime configuration in [AWS AppConfig](#) feature flags (or a comparable config service), combined with [deployment strategies](#) gated by CloudWatch alarms on latency and quality.

Per-class latency, token consumption, quality scores, and fallback-escalation rate belong on the same performance dashboards as total latency and reasoning iteration counts. A rising fallback rate on a class is an early indicator that the primary model no longer fits, either because traffic has shifted or because a newly available model would serve the class better. This typically shows up before the user-facing regression is large enough to alarm.

Implementation steps

- 1. Classify agent tasks by reasoning complexity:** Group the workload's tasks into classes based on the reasoning they require, for example, single-step extraction or classification, structured multi-step reasoning over known steps, and open-ended investigation. Document representative examples per class so routing decisions are auditable and can be revisited as task distributions shift. Use the classification as the input to model assignment rather than letting each caller pick a model case by case.
- 2. Benchmark candidate models on the workload's own task distribution:** Build an evaluation set representative of production traffic and score candidate models against it using [Amazon Bedrock AgentCore Evaluations](#) or an equivalent evaluation harness. Capture quality signals (correctness, goal success rate, tool-trajectory match) alongside latency and tokens so you can identify the smallest model meeting the quality bar for each class. Treat leaderboard rankings as a starting shortlist, not as the decision.
- 3. Map each task class to a model or model tier:** Define a small set of tiers, for example, fast, standard, and advanced, and assign each class to the tier that benchmarks demonstrate is sufficient. For each tier, select a specific model or Amazon Bedrock inference profile. Where routing within a family is the goal, [Amazon Bedrock intelligent prompt routing](#) can make the per-request pick automatically. For document-heavy tasks, consider [Amazon Bedrock Data Automation](#) as the right example rather than a general-purpose vision LLM.

4. **Implement routing logic that dispatches requests to the assigned model at runtime:** Resolve the task-class-to-model mapping at the start of each request and issue the inference call against the chosen model. When crossing providers or model families, a framework abstraction such as [Strands Agents model providers](#) keeps the routing code stable as providers change underneath it.
5. **Configure a cascading fallback to a more capable model for low-confidence or failing outputs:** Define structured signals, confidence score below a threshold, schema validation failure, parse error, or an explicit incomplete response, that escalate the specific request to a more capable model. Limit the escalation to a single step so tail latency stays predictable, and log both the primary and fallback decision for each request that escalates.
6. **Externalize model assignments and routing rules as runtime configuration:** Hold model identifiers, task-class-to-tier mappings, and fallback rules in [AWS AppConfig](#) feature flags (or an equivalent config service) and read them at request time. Decoupling selection from deployment lets new models be evaluated, promoted, or rolled back without redeploying the agent.
7. **Roll out model changes progressively with automated rollback:** Use [AWS AppConfig deployment strategies](#) to shift traffic to a new model in steps with bake-time validation, and attach CloudWatch alarms on latency and quality so the change rolls back automatically when the alarm fires. Treating a model swap as a monitored deployment makes frequent model updates safe.
8. **Emit per-task-class telemetry for latency, tokens, quality, and fallback rate:** Publish per-class metrics so the effect of each routing decision is visible on dashboards and can be alarmed. Use [Amazon CloudWatch generative AI observability](#) together with [AgentCore Observability](#) to attribute latency, tokens, and evaluation scores to the class and model that served each request.
9. **Review routing decisions against production telemetry on a defined cadence:** Schedule periodic reviews of per-class latency distributions, fallback-escalation rates, and quality scores, and re-run AgentCore Evaluations against newly released models using the same task-distribution benchmark. Promote, demote, or re-tier models based on observed data rather than provider release cadence.

Resources

Related best practices:

- [AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets](#)
- [AGENTPERF02-BP01 Design efficient reasoning pipelines](#)

- [AGENTPERF02-BP03 Optimize agent execution paths for reduced latency](#)
- [AGENTPERF02-BP04 Optimize streaming responses and time-to-first-token for agent interactions](#)

Related documents:

- [Understanding intelligent prompt routing in Amazon Bedrock](#)
- [Evaluate agent performance with Amazon Bedrock AgentCore Evaluations](#)
- [Amazon Bedrock Data Automation](#)
- [What is AWS AppConfig?](#)
- [Create a deployment strategy \(AWS AppConfig\)](#)
- [Amazon CloudWatch generative AI observability](#)
- [Foundations of agentic AI on AWS](#)
- [Economics for agentic AI on AWS](#)
- [Blog: Build reliable AI agents with Amazon Bedrock AgentCore Evaluations](#)
- [Strands Agents Model Providers](#)

Related videos:

- [AWS re:Invent 2025 - Mastering model choice: The 3-step Amazon Bedrock advantage \(AIM391\)](#)

Related examples:

- [GitHub: Amazon Bedrock AgentCore samples, Evaluations](#)

Related workshops:

- [Diving Deep into Bedrock AgentCore, Evaluations](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [AWS AppConfig](#)
- [Amazon CloudWatch](#)

AGENTPERF02-BP03 Optimize agent execution paths for reduced latency

Most of an agent request's total latency time is spent waiting on model inference, retrieval, tool invocations, and memory lookups rather than on CPU work inside the agent process. Executing independent operations concurrently, reusing warm connections and runtimes, and deduplicating repeated lookups within a single request cut total latency without changing models or prompts.

Desired outcome:

- You have independent operations within an agent request executed concurrently sequential execution is reserved for operations with genuine data dependencies.
- You have connections to downstream services, model endpoints, tool APIs, memory stores, vector indexes, pooled and reused across requests rather than reestablished per invocation.
- You have runtime cold starts removed from the critical path or bounded through provisioned capacity, pre-warming, or persistent execution environments.
- You have repeated lookups within a single request resolved from a request-scoped cache, so duplicate work isn't paid twice inside one invocation.

Common anti-patterns:

- Executing independent operations sequentially when they share no data dependencies, making total latency the sum of operation durations rather than the slowest operation.
- Establishing new connections to model endpoints, tool APIs, or data stores on every invocation, paying connection setup and TLS handshake costs on the critical path.
- Running agent code on compute that pays a cold-start penalty on the critical path without provisioned capacity, pre-warming, or a persistent runtime to absorb it.
- Re-executing the same lookup multiple times within a single request, for example, fetching the same knowledge base passage or user profile across consecutive reasoning steps, with no request-scoped cache to deduplicate the work.

- Introducing parallelism without respecting downstream rate limits or connection pool capacity, so concurrent calls throttle or queue and the intended latency win turns into added latency plus failures.

Benefits of establishing this best practice:

- Overlapping independent operations makes the critical path track the slowest operation rather than the sum of every operation.
- Amortizing connection setup and runtime initialization across requests avoids paying those costs on every invocation.
- Parallel calls that respect downstream capacity avoid the throttling and retry storms naive parallelism triggers.
- Overlapping I/O-bound waits, retrievals and tool calls, with compute-bound work such as inference and parsing keeps the agent productive during waits.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Most of an agent request's total latency time is spent waiting, on model inference, retrieval, tool invocations, and memory lookups, rather than on CPU work inside the agent process. The structure of how those waits are composed dominates total latency more than the speed of any single downstream dependency. Four structural decisions typically affect latency:

1. Running independent operations concurrently rather than sequentially
2. Reusing warm connections and runtimes across invocations
3. Removing cold starts from the critical path
4. Deduplicating repeated lookups within a single request.

Each decision is independent of model and prompt choices, so the gains compound with the reasoning-loop and model-selection optimizations addressed elsewhere in this pillar.

Concurrency is the largest impact when the agent fans out across independent data sources or tool calls. Dependency analysis identifies operations that share no data dependency (for example, a personalization lookup and a knowledge-base query issued from the same reasoning step, and executes them in parallel so the step's latency equals the slowest operation rather than the sum).

Agent frameworks expose this directly. Strands Agents executes independent tool calls emitted in a single reasoning step concurrently, and graph-based orchestrators such as LangGraph fan out across independent edges. The constraint is downstream capacity, where concurrent model calls and tool invocations must respect [Amazon Bedrock service quotas](#) (requests-per-minute and tokens-per-minute) and tool-API rate limits, or parallelism converts into throttling and retry storms that undo the latency win.

Connection reuse and cold-start removal address the per-invocation setup costs that compound with concurrency. HTTP connections to model endpoints and tool APIs should persist across invocations through the SDK or HTTP-client connection pool rather than be opened and torn down per request. Each fresh connection pays TLS-handshake and connection-setup overhead on the critical path that a pooled connection avoids entirely, and that overhead accumulates across fan-out and across invocations.

Database connections follow the same principle. [Amazon RDS Proxy](#) pools connections to Aurora and RDS so serverless agents don't exhaust database connection limits or pay connection-setup latency per invocation. At runtime, [Amazon Bedrock AgentCore Runtime sessions](#) reuse a dedicated microVM across invocations that share a session identifier, which removes cold starts while a session is active and preserves in-memory state across reasoning steps.

For agents hosted on AWS Lambda, [Lambda provisioned concurrency](#) pre-initializes execution environments and [Lambda SnapStart](#) restores from a cached snapshot on supported runtimes, reducing first-invocation latency to sub-second at the cost of continuous capacity or per-restoration charges.

Request-scoped caching addresses redundant work that happens inside a single invocation rather than across invocations. A reasoning loop that calls the same tool twice, retrieves the same passage across successive steps, or refetches the same user profile in the planner and the executor wastes latency budget on repeated I/O. A cache keyed by the request or session identifier deduplicates these lookups for the remainder of the request without the consistency complexity of a cross-request cache.

The scope is deliberately narrow, persistent and cross-request caches such as Amazon Bedrock prompt caching and semantic caches are higher-level optimizations addressed in context- and memory-focused best practices. However, request-scoped deduplication is frequently the lowest-risk caching optimization available, because the cache is discarded at the end of the invocation and can't serve stale data to a subsequent request.

Implementation steps

- 1. Profile the critical path to identify parallelizable operations:** Trace a representative sample of production requests with the performance telemetry already in place to decompose each invocation into per-operation durations and dependencies. Identify operations that share no data dependency, separate tool calls, independent retrievals, personalization lookups alongside knowledge queries, and flag the sequential segments where concurrency would collapse wall-clock latency onto the slowest operation. Revisit the inventory as prompts and tools change, because dependency graphs shift with them.
- 2. Execute independent operations concurrently within downstream capacity limits:** Configure the agent framework to fan out independent tool calls and retrievals in the same reasoning step, Strands Agents, LangGraph, and similar frameworks expose this as a native primitive. Bound concurrency to the downstream service's capacity, [Amazon Bedrock service quotas](#), tool-API rate limits, and database connection ceilings, so a step that fans out to 10 concurrent calls doesn't trigger throttling that costs more latency than it saves.
- 3. Reuse connections to model endpoints and external APIs across invocations:** Configure the HTTP client's connection pool so TLS sessions to Amazon Bedrock and tool APIs persist across invocations rather than being reestablished per request. On runtimes that preserve memory across invocations, [AWS Lambda execution environments](#), container-based services, and long-running services, initialize the client once per execution environment rather than per invocation so its connection pool survives across calls. A warm invocation should pay zero connection-setup latency on downstream calls.
- 4. Pool database connections through a managed connection pool:** For agents that read from or write to relational data, front Aurora and RDS databases with [Amazon RDS Proxy](#) so serverless invocations share a bounded pool of database connections rather than opening a new connection each time. Without a pooler, concurrent agent invocations exhaust the database's connection ceiling and pay per-invocation setup latency on the critical path, both failure modes worsen as parallelism increases.
- 5. Remove cold starts from the agent execution runtime:** Select a runtime that keeps the agent's execution environment warm on the critical path. [Amazon Bedrock AgentCore Runtime sessions](#) reuse a dedicated microVM across invocations that share a session identifier, preserving in-memory state and avoiding per-invocation cold starts while the session is active. For Lambda-hosted agents, [Lambda provisioned concurrency](#) pre-initializes execution environments and [Lambda SnapStart](#) restores from a cached snapshot on supported runtimes. Always-on container services such as Amazon ECS or Amazon EKS avoid cold starts entirely at the cost of continuous capacity. Choose based on traffic shape rather than runtime preference.

6. **Deduplicate repeated lookups within a single request using a request-scoped cache:** Add an in-memory cache scoped to the lifetime of the request or session that memoizes idempotent lookups, tool responses, retrieved passages, user-profile reads, keyed by input. A reasoning loop that calls the same tool twice or retrieves the same passage across successive steps resolves the second call from the cache, recovering that latency without the consistency complexity of a cross-invocation cache. The cache is discarded at the end of the request, so it can't serve stale data to a subsequent invocation.
7. **Re-measure the critical path after each structural change and as traffic grows:** After applying concurrency, pooling, cold-start, or caching changes, re-profile the critical path under representative production load to confirm the optimization held and did not introduce new failure modes such as throttling or connection-pool saturation. Repeat the measurement as traffic grows, because parallelism bounded correctly at launch frequently exceeds quotas at higher scale and the latency profile silently regresses before an SLO is exceeded.

Resources

Related best practices:

- [AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets](#)
- [AGENTPERF02-BP02 Implement task-appropriate model selection strategies](#)
- [AGENTPERF02-BP04 Optimize streaming responses and time-to-first-token for agent interactions](#)
- [AGENTPERF03-BP04 Establish efficient agent caching and data access patterns](#)

Related documents:

- [Amazon Bedrock service quotas](#)
- [Amazon Bedrock AgentCore Runtime sessions](#)
- [Amazon Bedrock latency-optimized inference](#)
- [Amazon RDS Proxy](#)
- [AWS Lambda execution environments](#)
- [AWS Lambda provisioned concurrency](#)
- [AWS Lambda SnapStart](#)
- [Building serverless architectures for agentic AI on AWS](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [AWS Lambda](#)
- [Amazon RDS Proxy](#)

AGENTPERF02-BP04 Optimize streaming responses and time-to-first-token for agent interactions

User-facing agents are judged on perceived latency, not total processing time. Time-to-first-token (TTFT), the delay before the first output reaches the user, is the dominant perceived-performance signal, and streaming delivery keeps TTFT short even when total processing takes several seconds. Agentic streaming is complicated by reasoning loops that must pause mid-stream to invoke tools and by multi-agent workflows where the final agent streams while upstream agents are still producing.

Desired outcome:

- You have TTFT tracked as a distinct KPI from end-to-end latency, with a target bounded by the interaction type.
- You have LLM output streamed to the user as it is generated, so the user begins seeing output well before the reasoning loop finishes.
- You have pre-inference latency, context assembly, prompt construction, retrieval, kept short enough that it doesn't dominate TTFT.
- You have tool invocations handled within streams so the user receives progress feedback rather than an unexplained pause when the agent calls a tool mid-response.
- You have multi-agent workflows designed so the user-facing agent begins streaming as soon as its inputs are available, rather than blocking until every upstream agent fully completes.

Common anti-patterns:

- Waiting for the complete agent response before delivering any output, making perceived latency equal to total processing time rather than time-to-first-token.
- Streaming the LLM inference call but not the pre-inference pipeline, so context retrieval and prompt construction add seconds of delay before the first token reaches the user.
- Pausing the output stream with no indication when the agent invokes a tool mid-response, so the user sees partial output followed by an unexplained pause.
- Blocking multi-agent workflows until every upstream agent finishes before the user-facing agent begins streaming, converting sequential coordination delay into user-visible latency.
- Treating TTFT as interchangeable with end-to-end latency in KPIs and alarms, so regressions in time-to-first-token go unnoticed while total-duration metrics look unchanged.

Benefits of establishing this best practice:

- Sub-second time-to-first-token keeps the agent feeling fast even when total processing time spans several seconds.
- Progress feedback replaces the unexplained pauses that would otherwise appear to the user as a stall when tools run mid-stream.
- Tracking TTFT as a distinct KPI surfaces drift in perceived responsiveness that end-to-end latency dashboards would otherwise hide.
- Progressive streaming in multi-agent workflows lets the user-facing agent deliver output concurrently with upstream processing rather than blocking until every upstream step completes.
- A short TTFT reduces the share of users who abandon interactive workloads before any output appears.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Time-to-first-token (TTFT) is typically the performance metric that most directly shapes user perception of an interactive agent. A response that begins within a few hundred milliseconds typically feels fast to users even when total generation takes several seconds. End-to-end latency and TTFT move independently. A faster model improves total duration but leaves TTFT unchanged when the pre-inference pipeline is the bottleneck, so tracking only total latency hides

the regressions users actually feel. The difference lies in instrumenting TTFT as a distinct metric, separate from total-duration dashboards.

Streaming the model's output is necessary but not sufficient: by the time the first token leaves the model, the agent can already have consumed the entire TTFT budget on work that happens before inference begins. Context assembly, prompt construction, retrieval, and serial pre-checks all count, and streaming recovers nothing from that window.

The pre-inference path is usually where the most significant TTFT improvements come from: compressing retrieval, narrowing retrieved context, and parallelizing independent pre-inference steps. The same concurrency and warm-connection patterns that reduce total latency elsewhere in this pillar apply to the pre-inference path, and they pay back specifically against TTFT. Post-inference filtering is also in the budget. [Amazon Bedrock Guardrails streaming modes](#) introduce an explicit trade-off between moderation accuracy and TTFT that must be tuned to the workload rather than left at default.

On Amazon Bedrock, [ConverseStream](#) is the model-agnostic streaming inference API recommended for chat and agent workloads, while [InvokeModelWithResponseStream](#) remains available when a model-specific payload shape is required. Both emit an event stream of content-block start, delta, and stop events that the agent layer translates into user-visible output.

Tool invocation introduces a discontinuity. When the model decides to call a tool, the event stream opens a content block of type `toolUse`, streams the tool's input as deltas, and then pauses while the agent runs the tool and feeds results back. A client that receives no signal during this gap shows the user partial output followed by a silent stall. The baseline pattern, buffer-and-resume with an explicit progress indicator, forwards a user-visible status the moment a tool-use block appears and resumes streaming when the next content block starts. More advanced patterns such as speculative streaming exist, but the baseline is that no silent pause reaches the user.

Multi-agent pipelines amplify the TTFT problem when every upstream agent must fully complete before the user-facing agent begins. Each serial handoff contributes its full duration to TTFT rather than overlapping with downstream work. Progressive streaming is the alternative, where the user-facing agent begins reasoning as soon as its minimum required inputs are available, and upstream agents' intermediate outputs stream into its context as they are produced.

Agent frameworks expose this pattern directly: [Strands Agents](#) yields agent events (tokens, tool calls, messages) as an async iterator that downstream consumers can subscribe to, and graph-based orchestrators such as LangGraph expose equivalent streaming primitives. Reserve

synchronous full-response handoff for workflows where the downstream agent genuinely can't begin until the upstream result is complete.

Two AWS capabilities reduce inter-token latency and coordination overhead beyond what API choice alone can deliver. [Amazon Bedrock latency-optimized inference](#) (in preview at publication) reduces inter-token latency on supported models through routing and capacity optimizations, at the cost of tighter throughput limits and model-specific token ceilings. For voice and real-time interactive workloads, [Amazon Bedrock AgentCore Runtime bi-directional streaming](#) over WebSocket or WebRTC allows the client to send input while the agent is still streaming output, the prerequisite for natural interrupt and turn-taking behavior in voice agents.

[Amazon Nova Sonic](#) provides a speech-to-speech path on Amazon Bedrock. You can route voice through Amazon Nova Sonic rather than separate speech-to-text, text-generation, and text-to-speech stages, collapsing multiple sequential stages into one bidirectional stream. This approach typically provides substantial TTFT improvements for voice workloads.

Implementation steps

- 1. Define TTFT targets for each user-facing interaction type:** Set a TTFT target per workload based on how the user consumes output, text chat tolerates hundreds of milliseconds, voice tolerates far less, batch pipelines have no user-facing TTFT at all. Treat the target as a budget to be allocated across pre-inference work, model first-token latency, and any post-processing, and anchor it to user research or published interaction-design norms rather than a round number.
- 2. Instrument TTFT as two distinct metrics, pipeline TTFT and model TTFT:** Emit *pipeline TTFT* at the user-facing boundary (the first output byte reaching the client) as the SLO KPI, and *model TTFT* at the first text delta from the inference call whose output first streams to the user, which isolates model and routing behavior from pre-inference and post-processing contributions. Publish both through [OpenTelemetry through the CloudWatch OTLP endpoint](#) or [CloudWatch Embedded Metric Format](#). When pipeline TTFT and model TTFT diverge, the gap points at the non-model contributors.
- 3. Extend TTFT instrumentation to multi-inference and voice workloads:** When an agent makes multiple inference calls per request, planners, routers, sub-agent fan-outs, treat silent upstream calls as part of the pre-inference budget and emit per-call TTFT as a tagged dimension so the contribution of each inference call remains visible for diagnosis. For voice workloads, add time-to-first-audio-chunk because text-token arrival is an upstream signal, not the user boundary.
- 4. Reduce pre-inference latency on the critical path to the first token:** Profile the work that happens between request arrival and the first inference call, context assembly, retrieval, prompt construction, pre-checks, and compress or parallelize it so most of the TTFT budget remains

when the model begins generating. The concurrency and connection-reuse patterns applied elsewhere to reduce end-to-end latency pay back specifically against TTFT when applied to the pre-inference path. Streaming token delivery can't recover any time lost before the first model call.

5. **Use streaming inference APIs rather than synchronous inference for user-facing agents:** Call [ConverseStream](#) as the model-agnostic default for chat and agent workloads. Use [InvokeModelWithResponseStream](#) only when a model-specific payload shape is required. Consume the event stream as it arrives and forward each content-block delta to the client rather than buffering the full response server-side.
6. **Tune Amazon Bedrock Guardrails streaming mode when guardrails are in the critical path:** If [Amazon Bedrock Guardrails](#) filters output on the user-facing path, choose the stream processing mode based on the workload's policy tolerance, synchronous processing raises moderation accuracy and raises TTFT, asynchronous processing preserves TTFT at the cost of potentially emitting a token that is later retracted. Drive the decision for mode by the content-risk profile of the workload.
7. **Surface tool-invocation events to the user rather than pausing the stream silently:** When the model emits a tool-use event mid-stream, forward a user-visible progress indicator to the client before the agent begins executing the tool, and resume streaming when the next content block starts. Use the explicit start and stop boundaries of the tool-use content block as the signal to transition the UI between streaming and working states, rather than letting the client see a silent gap.
8. **Stream multi-agent workflows progressively rather than blocking on upstream completion:** Design the orchestration so the user-facing agent begins reasoning as soon as its minimum required inputs are available, and pipe upstream intermediate events into its context as they are produced. Agent frameworks expose this streaming-handoff pattern directly through async-iterator primitives such as [Strands Agents'](#) streaming API and equivalent mechanisms in graph-based orchestrators. Reserve synchronous full-response handoff for workflows where the downstream agent genuinely can't begin until the upstream result is complete.
9. **Evaluate latency-optimized inference for inter-token latency on supported models:** [Amazon Bedrock latency-optimized inference](#) (in preview at publication) reduces inter-token latency on supported models, Amazon Nova Pro, Anthropic Claude 3.5 Haiku, and Meta Llama 3.1 70B/405B, through routing and capacity optimizations, at the cost of tighter throughput limits and model-specific token ceilings. Enable the latency mode on the runtime API and validate with the two-metric TTFT instrumentation that the reduction is real for the workload rather than a cache-warmed artifact.

- 10 Use AgentCore Runtime bi-directional streaming and Amazon Nova Sonic for voice and real-time workloads:** For voice agents and other workloads where the user must interrupt or turn-take, run the agent on [Amazon Bedrock AgentCore Runtime bi-directional streaming](#) over WebSocket or WebRTC so the client can send input while the agent is streaming output. Route voice specifically through [Amazon Nova Sonic](#), which collapses the separate speech-to-text, text-generation, and text-to-speech stages of a traditional voice pipeline into a single bidirectional stream, typically a substantial TTFT improvement for voice workloads.
- 11 Re-measure TTFT after each change and as traffic shifts:** Re-profile both pipeline TTFT and model TTFT under representative production load after applying streaming, pre-inference, tool-handling, or runtime changes, because optimizations that work in isolation frequently regress at scale. Alert on TTFT percentile violations distinct from end-to-end latency SLOs so regressions in perceived responsiveness surface before they reach users.

Resources

Related best practices:

- [AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets](#)
- [AGENTPERF02-BP02 Implement task-appropriate model selection strategies](#)
- [AGENTPERF02-BP03 Optimize agent execution paths for reduced latency](#)
- [AGENTPERF05-BP04 Implement efficient agent delegation and handoff patterns](#)

Related documents:

- [Amazon Bedrock ConverseStream API](#)
- [Amazon Bedrock InvokeModelWithResponseStream API](#)
- [Amazon Bedrock latency-optimized inference](#)
- [Amazon Bedrock Guardrails streaming](#)
- [Amazon Bedrock AgentCore Runtime bi-directional streaming](#)
- [Get started with bidirectional streaming using WebSocket on AgentCore Runtime](#)
- [Amazon Nova Sonic, real-time conversational speech](#)
- [CloudWatch Embedded Metric Format specification](#)
- [Publishing custom metrics to Amazon CloudWatch](#)

- [AWS blog: Bi-directional streaming for real-time agent interactions now available in Amazon Bedrock AgentCore Runtime](#)

Related examples:

- [Amazon Bedrock AgentCore samples, Runtime tutorials](#)
- [Amazon Bedrock AgentCore samples, Nova Sonic integration](#)

Related workshops:

- [Diving Deep into Bedrock AgentCore, Runtime](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Memory, context, and RAG optimization

Agentic AI systems depend on efficient access to contextual information through short-term conversational memory, long-term knowledge stores, retrieval-augmented generation pipelines, and LLM context windows to produce accurate, relevant responses. Memory and context management directly impacts both performance and quality. Overstuffing context windows increases inference latency and cost, and insufficient context leads to poor reasoning and hallucination. Optimizing this layer requires implementing tiered memory architectures that match storage and retrieval characteristics to access patterns, managing context windows to maximize information density without exceeding token limits, designing RAG pipelines that retrieve relevant information with minimal latency, and establishing caching strategies that reduce redundant retrievals. These optimizations matter because memory and context operations occur on every reasoning iteration, making their efficiency a multiplier across the agent lifecycle.

AGENTPERF03: How do you optimize memory management, context windows, and retrieval-augmented generation?

Capability intent

- Agent memory is organized into tiers that match storage technology to access pattern, so session reads and long-term semantic queries don't compete on the same hot path.
- Context windows are composed from discrete, budgeted components (system instructions, conversation history, retrieved knowledge, and tool schemas), with summarization, dynamic tool selection, and relevance-filtered retrieval keeping prompt size decoupled from session length.
- Retrieval-augmented generation pipelines combine semantic chunking, hybrid search, query transformation, and re-ranking so the agent receives precise context with sub-second latency on the first retrieval.
- Multi-layer caching (prompt prefixes, retrieval results, tool outputs) reduces redundant work across iterations and sessions, with TTLs and keys tuned per layer against measured hit rates and staleness tolerance.
- Retrieval is expressed as a set of agent-invoked tools with bounded iteration, sufficiency checks, and parallel sub-query execution so complex questions converge within the workload's latency budget.

Maturity levels

These levels summarize what each stage of maturity looks like for memory, context, and RAG optimization as a whole.

Level	Name	What it looks like
1	Initial	Memory, context, and retrieval are treated as single-tier concerns. Full conversation history and the entire tool catalog flow into every prompt, RAG chunks are fixed-size with no relevance

Level	Name	What it looks like
		filtering, and no caching exists beyond what individual services provide by default. Token consumption and retrieval latency are reviewed only after a cost spike or a user-reported slowdown.
2	Emerging	Teams have separated short-term from long-term memory and adopted basic summarization of conversation history. RAG is running through a managed service such as Amazon Bedrock Knowledge Bases , and prompts are structured so that system instructions and stable tool definitions sit in a reusable prefix. Per-component token usage and retrieval latency are measured for production agents but tuning is still manual and reactive.

Level	Name	What it looks like
3	Defined	Memory tiers use storage matched to access pattern, with Amazon Bedrock AgentCore Memory or Amazon ElastiCache serving short-term state and AgentCore Memory long-term strategies or Amazon OpenSearch Service serving cross-session knowledge. RAG pipelines use semantic chunking, hybrid search, and a re-ranking stage, with top-k caps and per-passage thresholds enforced. Prompts compose with Amazon Bedrock prompt caching, and prompt templates are versioned in Amazon Bedrock Prompt management with per-component token budgets.

Level	Name	What it looks like
4	Proactive	Retrieval is expressed as a set of agent tools through Amazon Bedrock AgentCore Gateway with sufficiency evaluation, bounded iteration , and parallel sub-query execution for complex questions. Multi-layer caching is active across prompt prefixes, semantic retrieval keys, and tool outputs, with per-layer hit rates monitored against floors. Memory extraction, eviction, and prompt-template promotion run through automated pipelines, and per-tier latency and cache hit rate feed continuous tuning in Amazon CloudWatch generative AI observability .

Level	Name	What it looks like
5	Optimized	Memory tier sizing, retrieval budgets, and cache TTLs are recalibrated continuously from production telemetry rather than by scheduled review. Prompt components, tool selection, and retrieval routing are evaluated through Amazon Bedrock AgentCore Evaluations gated in CI/CD, and optimization patterns flow back to internal communities of practice. The organization contributes memory and retrieval patterns to the broader agentic AI community.

Common issues to watch for

- Teams store all agent memory in a single database regardless of access pattern, which forces sub-millisecond session reads and large-scale semantic searches to compete on the same backend and degrades both.
- Conversation history and the full tool catalog are included in every prompt without summarization or dynamic selection, so prompt size and latency grow linearly with session length even after the information stops being useful.
- RAG pipelines rely on fixed-size chunking and raw top-k passages without a re-ranking stage or per-passage relevance thresholds, diluting useful context with low-signal chunks that crowd out higher-signal passages.
- Caching is either absent or applied at a single layer with a single TTL, missing the stacked savings available when prompt prefixes, semantic retrieval keys, and tool outputs are cached independently with data-type-specific policies.

- Agentic retrieval loops run without iteration caps, sufficiency evaluation, or parallel sub-query execution, which either cuts off complex questions prematurely or lets iterations grow unbounded and consume the workload's latency budget.

Best practices

- [AGENTPERF03-BP01 Implement tiered memory management systems](#)
- [AGENTPERF03-BP02 Optimize context window utilization and prompt management](#)
- [AGENTPERF03-BP03 Optimize RAG retrieval pipelines for latency and precision](#)
- [AGENTPERF03-BP04 Establish efficient agent caching and data access patterns](#)
- [AGENTPERF03-BP05 Implement agentic retrieval patterns for dynamic, agent-driven knowledge access](#)

AGENTPERF03-BP01 Implement tiered memory management systems

Agents that carry context across turns and sessions deliver more personalized and accurate responses, but only when memory retrieval doesn't become the latency bottleneck on every reasoning iteration. A tiered memory architecture separates fast, transient session state from durable cross-session knowledge so each tier's storage technology, access pattern, and lifecycle can be optimized independently rather than forced through a single store.

Desired outcome:

- You have agent memory separated into a short-term tier for in-session context and a long-term tier for cross-session knowledge, each backed by storage matched to its access pattern.
- You have automated lifecycle policies that extract durable insights from short-term memory into long-term strategies (semantic, episodic, summary, and user preference) and evict stale short-term state without manual intervention.
- You have per-tier retrieval latency tracked as a first-class KPI, with budgets that keep memory access from dominating the reasoning loop.
- You have long-term memory scoped and namespaced per user, session, or tenant so retrievals return only the records relevant to the current actor.

Common anti-patterns:

- Storing all agent memory in a single database regardless of access pattern, forcing sub-second session reads and large-scale semantic searches through the same storage layer.
- Persisting every turn of short-term memory indefinitely without extraction into long-term strategies or eviction, allowing session stores to grow without bounds and retrieval latency to degrade over time.
- Treating long-term memory as a single bag of records rather than differentiating between semantic facts, episodic events, conversation summaries, and user preferences, which forces every query to search all record types.
- Scoping long-term memory globally rather than per user, session, or tenant, so retrievals return cross-actor records that inflate context and leak information.
- Building custom tiered memory infrastructure from scratch instead of evaluating managed services that provide session stores, extraction strategies, and vector retrieval as primitives.

Benefits of establishing this best practice:

- Fast in-memory stores serve session reads in single-digit milliseconds while vector stores handle long-term semantic queries without coupling the two.
- Automated extraction and eviction policies keep each tier's footprint and retrieval latency stable as usage scales.
- Separating long-term memory into distinct strategies, semantic, episodic, summary, preference, lets the agent query only the record type relevant to its current reasoning step.
- Namespacing long-term memory by user, session, or tenant helps prevent cross-actor retrievals and keeps context relevant.
- Managed memory primitives remove the need to operate session stores, extraction pipelines, and vector indexes as bespoke infrastructure.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Memory access sits on the hot path of every reasoning iteration. An agent that reads session context and retrieves relevant long-term knowledge at every step pays that retrieval latency multiplied by the iteration count, which makes memory one of the largest use points in the reasoning loop.

The root cause of poor memory performance is typically an access-pattern mismatch. Using a single storage layer for both sub-millisecond session reads and large-scale semantic searches forces one pattern to carry cost and latency characteristics suited to the other. Tiering resolves the mismatch by splitting memory into a short-term tier for in-session context and a long-term tier for cross-session knowledge, then matching each tier to storage with the right latency, durability, and query model.

Short-term memory holds the turn-by-turn state an agent reads and writes within a single session: the last N turns, intermediate reasoning, tool outputs, and transient user-provided context.

[Amazon Bedrock AgentCore Memory](#) provides a managed short-term tier that stores session events and integrates with extraction into the long-term tier, removing the need to operate a separate session store or extraction pipeline.

For workloads that need sub-millisecond short-term reads or prefer to own the extraction pipeline, [Amazon ElastiCache \(Valkey\)](#) provides in-memory reads, TTL-based expiration, and native structures (hashes, lists, sorted sets) that map well to session data. Durability requirements for short-term memory are typically low, state can be regenerated or discarded on session end, so the tier should be sized for latency, not for archival.

Long-term memory holds durable knowledge that persists across sessions: user preferences, domain facts, past-interaction summaries, and episodic records of past task outcomes. Access is less frequent but operates over a much larger corpus and typically relies on semantic similarity rather than key lookup. AgentCore Memory provides a managed long-term tier with [four built-in strategies](#), semantic, episodic, summary, and user preference, each extracted from session events and indexed separately, so the agent can query only the store relevant to its current reasoning step.

For teams that prefer to own the long-term store directly, [agentic memory in Amazon OpenSearch Service](#) provides dense and hybrid retrieval over long-term records, and [Amazon Neptune Analytics](#) provides a graph-based alternative for domains where long-term memory is defined by relationships between entities, enabling multi-hop queries that vector similarity can't answer on its own. Separating strategies (managed or self-indexed) matters for performance, as every irrelevant record retrieved is latency and context budget spent on noise.

Tiers are only high-performing when their lifecycle is automated. Short-term state that isn't evicted grows until reads slow and session stores run out of memory, while long-term records that are not extracted from short-term events represent knowledge the agent has to relearn every session.

Managed services handle both movements: AgentCore Memory extracts long-term strategies from short-term events asynchronously and applies TTLs to short-term records, while self-managed stacks must build extraction and eviction explicitly, either by adopting an open source orchestration layer such as Mem0 or by writing bespoke pipelines on top of primitives like ElastiCache, OpenSearch, or Neptune Analytics.

Long-term memory must also be namespaced by actor (user, session, or tenant), because unscoped retrievals return records from other actors that inflate context and, depending on the deployment, leak information across isolation boundaries. Scoping is both a performance control (a smaller search space per query returns faster) and a correctness control.

Implementation steps

- 1. Inventory the memory the agent reads and writes:** List the distinct pieces of state the agent maintains, last-N-turn context, intermediate reasoning, tool outputs, user preferences, past-interaction summaries, domain facts, and for each note the access pattern (read per iteration, read per session, read per task class), retention requirement (session-scoped or durable), and query shape (key lookup or semantic search). This inventory is the input to tier selection, as without it, tier boundaries are drawn by guess and either fragment naturally grouped data or collapse patterns that should be separated. Record the inventory alongside the workload's performance budgets so tiering decisions can be audited and revisited.
- 2. Assign each inventoried item to a short-term or long-term tier:** Place session-scoped, high-frequency, latency-critical items in the short-term tier and durable, cross-session items queried semantically in the long-term tier. Avoid intermediate "working memory" tiers unless a concrete access pattern justifies one, most "working memory" is either active short-term state or a long-term record that has not been extracted. Document the tier boundary so every new memory item has a clear home.
- 3. Choose storage for each tier based on the tier's access pattern:** For the short-term tier, select [Amazon Bedrock AgentCore Memory](#) if you also want managed long-term extraction, or [Amazon ElastiCache \(Valkey\)](#) if you prefer to own the extraction pipeline. For the long-term tier, use AgentCore Memory's built-in strategies or [Amazon OpenSearch Service](#) for dense and hybrid retrieval. Resist using one storage layer for both tiers. It is the single most common cause of memory-bound latency regressions.
- 4. Configure long-term memory with strategies that match what the agent retrieves:** Enable the subset of [AgentCore's built-in long-term strategies](#), semantic, episodic, summary, and user preference, that correspond to the retrieval patterns in the inventory. Each strategy extracts a different shape of record from short-term events and indexes it separately, so the agent can

- query only the store relevant to its current step. In self-managed stacks, create equivalent per-strategy indexes rather than a single general-purpose corpus.
5. **Namespace every memory record to its actor (user, session, or tenant):** Attach an actor identifier to every short-term and long-term record and filter every retrieval by that identifier using AgentCore Memory's [actor and session scoping](#) or an equivalent filter in self-managed stacks. Scoping reduces the search space (lower retrieval latency) and helps prevent cross-actor context leakage (correctness and isolation). Align the actor key with the authentication identity used by the agent so scoping can't be bypassed by a missing filter in application code.
 6. **Automate extraction from short-term to long-term and eviction of stale short-term state:** Configure the managed extraction pipeline, AgentCore Memory's asynchronous strategy extraction, or build an equivalent job in self-managed stacks that reads session events, derives long-term records per enabled strategy, and writes them to the long-term index. Apply TTLs or sliding-window eviction to short-term state so session stores don't grow without bounds. Both movements must run without manual intervention. If either requires human action, memory growth and extraction lag will exceed design targets.
 7. **Emit per-tier retrieval latency as a first-class performance metric and set budgets:** Publish short-term read latency and long-term query latency as distinct time-series through [Amazon CloudWatch generative AI observability](#) or an equivalent pipeline, alongside tier size, hit rate, and extraction lag. Allocate each tier an explicit portion of the per-iteration latency budget so memory can't silently consume time reserved for inference or tool calls. Treat per-tier latency as an early indicator: sustained growth in long-term query latency usually signals index size or scope drift before it registers on end-to-end metrics.
 8. **Review tier sizing, strategies, and budgets against production telemetry on a defined cadence:** Schedule reviews of short-term tier size distributions, long-term strategy growth rates, extraction lag, and per-tier latency against budget. Tighten TTLs on short-term stores that are consistently oversized, disable long-term strategies that are never queried, and re-scope memory if retrievals are returning more cross-actor records than the scope intended. Tiering parameters set at launch rarely match production traffic unless they are reviewed on an ongoing basis.

Resources

Related best practices:

- [AGENTPERF03-BP02 Optimize context window utilization and prompt management](#)
- [AGENTPERF03-BP03 Optimize RAG retrieval pipelines for latency and precision](#)

- [AGENTPERF03-BP04 Establish efficient agent caching and data access patterns](#)
- [AGENTPERF03-BP05 Implement agentic retrieval patterns for dynamic, agent-driven knowledge access](#)

Related documents:

- [Amazon Bedrock AgentCore Memory](#)
- [AgentCore Memory, memory organization](#)
- [AgentCore Memory, long-term built-in strategies](#)
- [Agentic memory in Amazon OpenSearch Service](#)
- [Why use ElastiCache for agentic memory](#)
- [Foundations of agentic AI on AWS](#)
- [Blog: Amazon Bedrock AgentCore Memory, Building context-aware agents](#)
- [Blog: Building smarter AI agents, AgentCore long-term memory deep dive](#)
- [Blog: Build agents to learn from experiences using AgentCore episodic memory](#)
- [Blog: Build persistent memory for agentic AI applications with Mem0, ElastiCache for Valkey, and Neptune Analytics](#)

Related videos:

- [AWS re:Invent 2024 - Make agents remember with Amazon Bedrock AgentCore Memory \(AIM331\)](#)
- [AgentCore Deep Dive: Memory](#)
- [Solving LLM Amnesia: Cross Session Memory](#)

Related examples:

- [GitHub: Amazon Bedrock AgentCore samples, Memory tutorials](#)

Related workshops:

- [Diving Deep into Bedrock AgentCore, Memory](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon ElastiCache](#)
- [Amazon OpenSearch Service](#)
- [Amazon Neptune Analytics](#)

AGENTPERF03-BP02 Optimize context window utilization and prompt management

Every token sent to an LLM competes for the model's attention, consumes input-token cost, and adds inference latency, which makes prompt content a first-order performance lever. Effective context window management budgets tokens across prompt components, assembles only what the current task needs, and compresses or summarizes the rest. Without this discipline, conversation history and tool schemas crowd out reasoning capacity and inflate latency on every iteration.

Desired outcome:

- You have an explicit token budget allocated across prompt components, system instructions, conversation history, retrieved knowledge, and tool schemas, with per-component token usage measured on every request.
- You have context assembled dynamically per request, including only the tool definitions, retrieved passages, and conversation context relevant to the current task rather than a fixed maximal payload.
- You have conversation history bounded by summarization, sliding windows, or semantic compression so prompt size doesn't grow linearly with session length.
- You have prompt templates versioned and evaluated so changes to wording, ordering, or component composition are measured against quality and token-cost baselines before rollout.

Common anti-patterns:

- Including the full conversation history in every prompt without summarization or truncation, causing prompt size and inference latency to grow linearly with session length.
- Injecting the full tool catalog in every prompt regardless of task relevance, spending context budget on schemas the agent will not invoke for the current request.
- Passing RAG retrievals straight into the prompt without per-passage filtering or truncation, so low-relevance chunks displace more useful context.

- Treating the prompt as a single opaque string rather than a composition of components, making it impossible to attribute token consumption to system instructions, history, retrievals, or tool schemas.
- Shipping prompt changes without versioning or side-by-side evaluation, so quality or token-cost regressions are detected only after rollout.

Benefits of establishing this best practice:

- Removing tokens from the prompt can reduce request cost and shorten time-to-first-token on each iteration.
- Pruned, high-density context leaves more of the model's effective attention on the current task instead of parsing stale history or unused tool schemas.
- Summarization and sliding-window strategies decouple prompt size from conversation length, keeping per-turn latency and cost stable.
- Stable, component-structured prompts with invariant prefixes compose with prompt caching, turning a large portion of input tokens into cached reads at a fraction of standard input cost.
- Versioned templates with evaluation gates let prompt changes ship with quality and cost evidence rather than by guess.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Prompts behave as compositions of discrete components, system instructions, tool schemas, retrieved knowledge, conversation history, and the current user turn. Treating them as a single opaque string makes runaway growth impossible to attribute. Per-component token attribution on every request demystifies issues, such as a tool catalog that doubled when a new tool was registered. Component-level budgets that sum to the workload's input-length target make the trade-offs explicit: more budget for retrievals means less for history, and every reallocation becomes a deliberate decision.

The [Amazon Bedrock prompt design guidance](#) describes how clear instructions, output indicators, and question placement at the end of the prompt shape the system-instruction and user-turn components individually.

The cheapest tokens are the ones never sent. Most agents register many more tools than any single turn will invoke, so injecting the full tool catalog on every request spends context budget

on schemas the model ignores. [Amazon Bedrock AgentCore Gateway](#) addresses this with semantic search that returns only the tools relevant to the current user intent.

Retrieval-augmented context suffers from the same failure mode, passing raw top-K passages into the prompt without a relevance threshold or per-passage cap lets low-signal chunks displace higher-signal context.

Conversation history is the third inflation source: the [SummaryMemoryStrategy in Amazon Bedrock AgentCore Memory](#) maintains a running per-session summary that replaces raw turns after the session exceeds a threshold, decoupling prompt size from session length so per-turn latency remains stable as conversations grow.

Static prefixes unlock large, recurring input-token savings. [Amazon Bedrock prompt caching](#) accepts cache checkpoints in the system, tools, and messages fields on the Converse API, with cache reads charged at a reduced rate on a 5-minute default TTL, or a 1-hour TTL on Claude Opus 4.5, Haiku 4.5, and Sonnet 4.5 for longer-running or intermittent agent traffic.

Invariant content (system instructions, stable tool definitions, and pinned context) belongs before the checkpoint, and variable content (current user turn and session-specific retrievals) belongs after it so the cached prefix remains identical across turns. Simplified cache management for Claude models looks back approximately 20 content blocks for the longest matching prefix, so the most-reused content should sit within that range.

Prompts behave like code, so a change to wording, component order, or a tool description can shift quality, token count, and cache-hit rate simultaneously, while the effects remain invisible until rollout unless they are measured. [Amazon Bedrock Prompt management](#) stores prompts as versioned artifacts with variables, inference configuration, and optional prompt-caching settings, and its console variant comparison surfaces quality and token differences side by side before promotion.

[Prompt Optimization](#) generates model-specific rewrites that serve as candidate variants rather than drop-in replacements, and quality plus token-cost benchmarks on representative test sets should gate promotion of any variant.

Implementation steps

- 1. Define the prompt component taxonomy and per-component token budget:** Decompose every prompt into a fixed set of components, system instructions, tool schemas, retrieved knowledge, conversation history, and the current user turn, and allocate a token budget to each

- that sums to the input-length target derived from the workload's latency and cost SLO. Apply the [Amazon Bedrock prompt design guidance](#) to keep the system-instruction component clear, concise, and consistent with the user query placed at the end of the prompt.
2. **Instrument per-component token usage on every request:** Emit token counts for each component as structured logs or CloudWatch metrics alongside the inputTokens and outputTokens returned by the [Amazon Bedrock Converse API](#), dimensioned by agent ID and task type. Include cacheReadInputTokens and cacheWriteInputTokens from the Converse response so cache effectiveness is measured alongside component growth, and alert when any component trends outside its budget before it impacts the end-to-end SLO.
 3. **Assemble tool schemas dynamically with just-in-time selection:** Replace full-catalog injection with task-relevant selection so the prompt carries only the tools the agent might invoke for the current turn. For agents using [Amazon Bedrock AgentCore Gateway](#), enable semantic search by setting "searchType": "SEMANTIC" on the mcp protocol in the gateway's protocolConfiguration so tool retrieval narrows on user intent. Keep each tool's schema compact with clear parameter descriptions and required fields to reduce its per-invocation token weight.
 4. **Bound conversation history with summarization and a sliding window:** Cap raw-turn retention and replace older turns with a running summary after the session exceeds a threshold expressed in turns or tokens. Configure the [SummaryMemoryStrategy in Amazon Bedrock AgentCore Memory](#) with a namespaceTemplates entry such as /summaries/{actorId}/{sessionId}/ and inject the summary rather than the raw transcript into the prompt, paired with a small sliding window of recent turns to preserve near-term conversational detail.
 5. **Filter and truncate retrieved passages before prompt assembly:** Apply a relevance threshold, a per-passage token cap, and a total retrieval budget so low-signal chunks can't displace higher-signal context. Rank passages by relevance score, drop any below the threshold, and truncate long passages to the per-passage cap before composing the retrieved-knowledge component.
 6. **Structure prompts so they compose with prompt caching:** Order every prompt so invariant content (system instructions, stable tool definitions, pinned reference material) appears before variant content (user turn, per-request retrievals), then place [Amazon Bedrock prompt cache checkpoints](#) in the system, tools, and messages fields at the boundary between static and dynamic content. For long-running or intermittent sessions on supported Claude models, set "ttl": "1h" on the cachePoint to extend the cache window beyond the 5-minute default, keeping the 1-hour entry ahead of any 5-minute entries in the same request.
 7. **Version prompts and gate changes on evaluation:** Store prompts as versioned artifacts in [Amazon Bedrock Prompt management](#) with variables for dynamic inputs and inference configuration tied to the version, then use [Create version](#) to snapshot known-good drafts. Use

the side-by-side variant comparison in the prompt builder to evaluate candidates against each other, and promote a new version only after it clears quality and token-cost baselines on a representative test set.

8. **Generate model-tuned rewrites with prompt optimization:** Run candidate prompts through [Amazon Bedrock Prompt Optimization](#) to produce model-specific rewrites, passing the target model through `targetModelId` when using the `OptimizePrompt` API. Treat the optimized output as a candidate variant rather than a drop-in replacement, and compare it against the original on the same evaluation set so token-count reductions are weighed against any quality impact before promotion.

Resources

Related best practices:

- [AGENTPERF03-BP01 Implement tiered memory management systems](#)
- [AGENTPERF03-BP03 Optimize RAG retrieval pipelines for latency and precision](#)
- [AGENTPERF03-BP04 Establish efficient agent caching and data access patterns](#)
- [AGENTPERF06-BP01 Design optimized tool integration strategies](#)

Related documents:

- [Amazon Bedrock, Prompt caching for faster model inference](#)
- [Amazon Bedrock, Create a prompt using Prompt management](#)
- [Amazon Bedrock, Create a version of a prompt in Prompt management](#)
- [Amazon Bedrock, Optimize a prompt](#)
- [Amazon Bedrock, Design a prompt](#)
- [AgentCore Gateway, Performance optimization \(refined tool schemas and semantic search\)](#)
- [AgentCore Memory, Long-term built-in strategies \(semantic, episodic, summary, user preference\)](#)
- [Foundations of agentic AI on AWS](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)

AGENTPERF03-BP03 Optimize RAG retrieval pipelines for latency and precision

Retrieval-augmented generation gives agents access to knowledge beyond the model's training data, but every reasoning iteration that queries a retrieval pipeline pays its latency and inherits the quality of its results. A well-tuned RAG pipeline returns high-relevance passages within a small latency budget. Without this discipline, retrieval either dominates per-iteration latency or returns noisy context that degrades reasoning quality.

Desired outcome:

- You have a chunking strategy matched to the structure and query shape of each source corpus, with chunk size and boundaries tuned against retrieval precision rather than defaulted to a single fixed size.
- You have retrieval latency tracked per stage (embedding, search, and re-ranking) with explicit budgets so retrieval can't silently consume time allocated to reasoning or tool calls.
- You have hybrid retrieval and re-ranking used where the corpus and query mix justify their added latency, rather than stacked by default.
- You have query reformulation ahead of every retrieval, so recall holds when agent phrasing diverges from corpus vocabulary.
- You have retrieval precision and relevance continually evaluated against a representative query set so quality regressions are caught before they reach production behavior.

Common anti-patterns:

- Using a single fixed chunk size across all document types, forcing structured content (tables, code, lists) through the same boundaries as flowing prose and splitting related information across chunks.
- Passing raw top-K retrieval results to the LLM without re-ranking or relevance filtering, letting low-signal passages displace high-signal context in the agent's prompt.
- Embedding the agent's raw query without reformulation, missing relevant documents that use different terminology than the query phrasing.
- Running every retrieval through pure dense similarity search when the corpus contains exact identifiers, code, or numeric values that keyword search recovers more reliably.

- Choosing an embedding model once and never revisiting it, missing precision gains from newer models or domain-tuned alternatives.
- Treating retrieval latency as a single metric rather than attributing it across embedding, search, and re-ranking stages, so performance regressions can't be localized.

Benefits of establishing this best practice:

- Stage-level attribution and index tuning keep embedding, search, and re-ranking within a predictable per-retrieval budget.
- Matching chunking strategy to document structure, hybrid retrieval where warranted, and re-ranking before context delivery keeps noise out of the prompt.
- High-precision first retrievals reduce the number of reasoning iterations that retry retrieval with reformulated queries.
- Continuous evaluation against a representative query set detects relevance drift before it reaches production.
- Tighter, more relevant retrievals consume less of the context window, leaving more budget for reasoning.

Level of risk exposed if this best practice is not established: High

Implementation guidance

At ingest, four decisions fix what any future retrieval can see before a single query runs: parsing, chunking, embedding, and the choice of [vector store](#). Query-time stages can filter and reorder what ingest produced, but they can't recover information ingest discarded. This asymmetry is the architectural reason RAG pipelines can't be tuned as a single knob. Errors that are built in at ingest require re-ingesting the whole corpus to fix. Query-time misconfigurations can be patched without touching storage. When designers treat RAG as an unknown, they inherit both sides of that commitment without seeing where it was made.

Chunking is an ingest-time commitment with no cheap fix. Fixed-size chunks fragment tables and mix unrelated passages when topics shift mid-chunk. Hierarchical chunking preserves nested-document relationships but roughly doubles the index footprint because parent chunks are indexed alongside their children. Semantic chunking breaks on meaning rather than token count and can disagree with domain experts about where a topic actually shifts. [Advanced parsing](#) recovers figures and tables from PDFs before chunking runs but adds a foundation-model invocation per

document. Getting any of these wrong isn't a query-time tuning problem, requiring a full re-embedding of the corpus to fix.

Three query-time stages each convert latency into precision, and the embedding model used at ingest sets the floor for all three.

- Hybrid retrieval adds BM25 scoring alongside vector similarity, recovering exact-match queries but doubling first-stage work.
- Re-ranking runs a second, heavier model over a broad top-k to earn back prompt tokens, spending milliseconds per passage.
- Query reformulation expands a single query into several, trading fan-out latency for recall when agent phrasing misses corpus terminology.

Stacking these stages doesn't give additive precision gains: rerank over hybrid often matches rerank over dense-only, and reformulation paired with re-ranking can overlap in what each fixes. Layering all three yields a precision return that flattens before the last layer contributes, and a latency cost that doesn't.

Between one deploy and the next, corpora grow, embedding models update, re-rankers retrain, and agent query patterns shift. Each change can move retrieval quality in either direction with no in-band signal. Drift is only detectable against a representative query set labeled with expected passages.

Recall@k and nDCG@k quantify whether the right passages were returned, and the RAG triad (context relevance, answer relevance, groundedness) extends the measurement to whether retrieved context actually supported the answer. The eval set is also the joint between corpus owners, who update documents, and pipeline owners, who tune stages. Without shared evaluation, teams can ship regressions that other teams aren't aware of.

Implementation steps

1. **Inventory source corpora and query patterns:** For each knowledge corpus the agent will query, record document type (prose, hierarchical document, PDF with figures, structured tables, code), typical query shape (conceptual intent vs. exact identifier), expected query volume, and precision/latency budget. This inventory drives every downstream choice (chunking strategy, embedding model, index configuration, and re-ranker placement). Without it, the pipeline is tuned by guess against an imagined average.

2. **Choose a chunking and parsing strategy per corpus:** For flowing prose, use semantic chunking to break on meaning. For nested documents, use hierarchical chunking to preserve parent-child context. For PDFs and multimodal content with figures or tables, enable [advanced parsing](#) via a foundation model or Amazon Bedrock Data Automation before chunking. Configure the strategy in the [Knowledge Base chunking configuration](#) per data source rather than defaulting every source to fixed-size chunks.
3. **Select an embedding model and dimensionality:** Pick an embedding model that covers the modalities, languages, and domain of the corpora and whose cost and latency fit the per-retrieval budget. [Amazon Titan Text Embeddings V2](#) exposes configurable output dimensions so text-corpus index size, recall, and query latency can be balanced against each other, and [Amazon Nova Multimodal Embeddings](#) produces a single embedding space across text, images, video, and audio for corpora that contain mixed modalities. Re-evaluate the choice when new models are released, because embedding quality improvements translate directly into retrieval precision.
4. **Configure the index for the retrieval pattern:** For corpora with both conceptual and exact-match queries, enable hybrid search rather than relying on pure dense similarity. With Amazon Bedrock Knowledge Bases backed by Amazon OpenSearch Service Serverless, set [overrideSearchType: HYBRID](#) on the Retrieve request to combine vector and raw-text scoring in a single call. For direct OpenSearch workloads, configure [neural and hybrid search](#) explicitly. Apply metadata filters on the retrieve request to narrow the search space by document source, freshness, or scope before vector similarity runs, and tune [vector index parameters](#) against retrieval precision on a representative query set so the index favors recall or latency according to the workload's budget.
5. **Add re-ranking before context delivery:** Run first-stage retrieval at a wider top-K than the prompt context budget allows, then pass the results through [Amazon Bedrock Rerank](#) to produce a higher-precision subset. The re-ranker compensates for vector-search noise and lets the LLM receive fewer but higher-relevance passages, which both tightens prompt quality and reduces input tokens. Configure the re-ranker on the Knowledge Base retrieve request rather than orchestrating it client-side so the hop stays inside the retrieval path.
6. **Enable query reformulation:** Transform agent-generated queries before they hit the index so retrieval doesn't fail on terminology mismatches with the source corpus. Knowledge Bases supports [query reformulation and decomposition](#) through the Retrieve API, expanding a broad question into focused sub-queries executed against the index. Prefer the managed reformulation path over a bespoke preprocessing step so it stays colocate with the retrieval hop and benefits from future improvements to the feature.

7. **Instrument per-stage retrieval latency and relevance:** Emit distinct metrics for embedding latency, search latency, re-ranker latency, top-k size, and the relevance score distribution of returned passages, dimensioned by data source and query class. Per-stage attribution makes it possible to localize regressions. A rising re-ranker latency with a stable search latency points at a different root cause than the reverse. Set per-stage budgets that sum to the pipeline's end-to-end latency target so any single stage exceeding its budget alerts before end-to-end performance is affected.
8. **Evaluate the pipeline on a representative query set on a defined cadence:** Maintain a fixed evaluation set that spans the corpora and query shapes in the inventory, and run it against the pipeline on every significant change (new data source, chunking change, embedding or re-ranker upgrade, index parameter tuning), following the approach in [Evaluate and improve performance of Amazon Bedrock Knowledge Bases](#). Track Recall@k, nDCG@k, and RAG triad scores per change so quality regressions are caught before rollout, and refresh the evaluation set as real query patterns drift.

Resources

Related best practices:

- [AGENTPERF03-BP01 Implement tiered memory management systems](#)
- [AGENTPERF03-BP02 Optimize context window utilization and prompt management](#)
- [AGENTPERF03-BP04 Establish efficient agent caching and data access patterns](#)
- [AGENTPERF03-BP05 Implement agentic retrieval patterns for dynamic, agent-driven knowledge access](#)

Related documents:

- [Amazon Bedrock Knowledge Bases, How content chunking works](#)
- [Amazon Bedrock Knowledge Bases, Parsing options for your data source](#)
- [Amazon Bedrock Knowledge Bases, Query configurations](#)
- [Amazon Bedrock, Improve the relevance of query responses with a re-ranker model](#)
- [Amazon Titan Text Embeddings models](#)
- [Amazon Nova Multimodal Embeddings](#)
- [Amazon OpenSearch Service, Vector search](#)

- [Amazon OpenSearch Service Serverless, Configure Neural Search and Hybrid Search](#)
- [Blog: Amazon Bedrock Knowledge Bases, advanced parsing, chunking, and query reformulation](#)
- [Blog: Evaluate and improve performance of Amazon Bedrock Knowledge Bases](#)
- [Choosing an AWS vector database for RAG use cases](#)

Related videos:

- [AWS re:Invent 2025 - Advanced agentic RAG Systems: Deep dive with Amazon Bedrock \(AIM425\)](#)

Related examples:

- [GitHub: Amazon Bedrock samples, Knowledge Bases and RAG](#)

Related tools:

- [Amazon Bedrock Knowledge Bases](#)
- [Amazon OpenSearch Service](#)
- [Amazon CloudWatch](#)

AGENTPERF03-BP04 Establish efficient agent caching and data access patterns

Agents that repeatedly fetch the same data benefit from caching, breaking the cycle of redundant retrievals speeds up every reasoning iteration. Agentic workloads often access the same tool outputs, retrieved documents, computed embeddings, and configuration data across multiple reasoning iterations, sessions, or agents in a multi-agent workflow. Without caching, each access pays the full latency and cost of the original operation.

Desired outcome:

- You have multi-layer caching that removes redundant computations and data fetches across reasoning iterations, sessions, and agents.
- You have cache hit rates monitored and optimized.
- You have cache invalidation policies tuned to balance freshness requirements with performance benefits.

Common anti-patterns:

- Implementing no caching at all, forcing agents to re-fetch the same documents, re-compute the same embeddings, and re-invoke the same tools on every reasoning iteration.
- Using a single cache TTL for all data types without considering freshness requirements, producing either stale data (TTL too long) or poor hit rates (TTL too short).
- Designing cache keys based only on exact string matching, missing cache hits for semantically equivalent queries that use different phrasing.

Benefits of establishing this best practice:

- Cache hits substantially reduce latency for repeated data access.
- Removing redundant LLM inference calls and external API invocations lowers cost.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Caching is applied at multiple layers of the agent stack, and each layer has its own invalidation discipline.

At the LLM inference layer, [Amazon Bedrock prompt caching](#) caches and reuses common prompt prefixes (like system instructions and tool definitions) across invocations, reducing both latency and cost for repeated portions of prompts, prompt caching savings compound further when combined with Amazon Bedrock's Flex pricing tier for development and testing workloads.

At the retrieval layer, caching RAG query results under semantic cache keys (embedding-based similarity) rather than exact string matching lets semantically similar queries share cached results.

At the tool invocation layer, caching tool outputs based on input parameters with TTLs matched to the data's freshness requirements, a cached stock price has a very different TTL than a cached company description.

Cache warming is valuable where access patterns are predictable. If agents frequently access the same knowledge base sections during business hours, pre-warming the cache before peak periods avoids the first-miss penalty for early users. Data access patterns benefit from batching: retrieving multiple items in a single round trip rather than making sequential individual requests reduces both latency and connection overhead.

Monitoring cache hit rates, latency savings, and cost savings per cache layer in Amazon CloudWatch makes caching a tunable parameter.

Implementation steps

1. **Identify cacheable data across the agent stack:** Enumerate LLM prompt prefixes, RAG results, tool outputs, session state, and configuration data, each has its own access pattern, freshness requirement, and cache layer.
2. **Enable Amazon Bedrock prompt caching for common prompt prefixes shared across invocations:** Turn on [Amazon Bedrock prompt caching](#) and structure prompts so system instructions and tool definitions appear before variable content, letting the cached prefix be reused across requests.
3. **Implement retrieval result caching with semantic cache keys and data-type-specific TTLs:** Cache RAG results under embedding-based similarity keys so semantically equivalent queries share results, and tune TTLs to the freshness needs of each data type rather than applying a single global TTL.
4. **Implement tool output caching with TTLs calibrated to data freshness requirements:** Cache tool outputs under input-parameter keys with TTLs that match how fast each tool's data changes, short TTLs for real-time data, long TTLs for static reference data.
5. **Monitor cache hit rates and latency savings per cache layer using CloudWatch:** Publish hit rate, miss rate, and latency savings per cache layer as CloudWatch metrics so TTLs and warming strategies can be tuned from data rather than assumption.

Resources

Related best practices:

- [AGENTPERF03-BP01 Implement tiered memory management systems](#)
- [AGENTPERF03-BP03 Optimize RAG retrieval pipelines for latency and precision](#)
- [AGENTPERF02-BP03 Optimize agent execution paths for reduced latency](#)

Related documents:

- [Amazon Bedrock prompt caching](#)
- [Blog: Effectively use prompt caching on Amazon Bedrock](#)
- [Blog: Optimize LLM response costs and latency with effective caching](#)

- [Foundations of agentic AI on AWS](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon ElastiCache](#)
- [Amazon DynamoDB](#)
- [Amazon CloudWatch](#)

AGENTPERF03-BP05 Implement agentic retrieval patterns for dynamic, agent-driven knowledge access

Complex questions often require information from multiple sources, iterative refinement, or real-time data that a single retrieval pass can't provide. In agentic retrieval the agent actively controls the retrieval process as part of its reasoning loop, deciding when to retrieve, what to retrieve, which retrieval tool to use, and whether the retrieved context is sufficient before proceeding. Each iteration adds embedding generation, vector search, re-ranking, and context injection overhead, so the retrieval loop needs explicit termination conditions.

Desired outcome:

- You have agents retrieving the right information in the minimum number of iterations required.
- You have simple questions answered with a single retrieval and complex questions handled through structured multi-hop retrieval with explicit termination conditions.
- You have the agent selecting the most appropriate retrieval tool for each query type.
- You have retrieval iteration counts, per-iteration latency, and sufficiency rates tracked and optimized.

Common anti-patterns:

- Treating all retrieval as a single-shot preprocessing step, forcing the agent to work with whatever context was retrieved on the first attempt regardless of sufficiency.
- Allowing agents to retrieve iteratively without retrieval budgets or termination conditions, producing unbounded retrieval loops that consume tokens and latency without converging.

- Routing all retrieval through a single pipeline regardless of query type, missing opportunities to use faster or more appropriate retrieval tools for different information needs.

Benefits of establishing this best practice:

- Parallel sub-query execution and retrieval-tool routing reduce end-to-end latency by selecting the fastest appropriate source.
- Explicit budgets that cap iterations and total tokens keep retrieval costs under control.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Design retrieval as a set of agent tools rather than a monolithic pipeline. Distinct retrieval tools for different knowledge access patterns let the agent route to the right source:

- A semantic search tool backed by [Amazon Bedrock Knowledge Bases](#) for conceptual questions
- A structured query tool for exact lookups by identifier
- A real-time data tool for information requiring current values
- A web search tool for questions beyond the organization's knowledge base
- A document processing tool backed by [Amazon Bedrock Data Automation](#) for extracting structured data from images, forms, and tables

[Amazon Bedrock AgentCore Gateway](#) exposes retrieval tools as MCP-compatible endpoints, and registering each tool with clear descriptions, what question types it handles, what data sources it accesses, and its expected latency guides the agent's tool selection.

Retrieval sufficiency evaluation is a lightweight assessment after each retrieval iteration, typically run by a smaller, faster model. The evaluator judges whether the retrieved context is sufficient, identifies gaps, and formulates refined queries. A maximum retrieval iteration limit (typically 2-3 iterations) helps prevent unbounded loops. If the agent has not retrieved sufficient context within the budget, it proceeds with the best available context and communicates uncertainty.

For complex questions requiring multiple sources, query decomposition breaks the question into focused sub-queries and runs independent sub-queries concurrently. Per-task retrieval performance budgets, derived from the task's overall latency SLO, keep the iterative pattern inside the workload's target.

Implementation steps

1. **Implement distinct retrieval tools for different knowledge access patterns:** Register a semantic search tool, a structured-query tool, a real-time data tool, a web search tool, and a document processing tool through [Amazon Bedrock AgentCore Gateway](#) with clear descriptions that guide the agent's tool selection.
2. **Implement retrieval sufficiency evaluation as a lightweight post-retrieval assessment:** Use a small, fast model to judge whether retrieved context is sufficient, identify gaps, and formulate refined queries for the next iteration.
3. **Configure maximum retrieval iteration limits with graceful fallback to best-available context:** Cap iterations at 2-3 for most tasks, and when the budget is exhausted proceed with the best context obtained and communicate uncertainty rather than looping without bounds.
4. **Implement query decomposition for complex questions, running independent sub-queries concurrently:** Break multi-source questions into focused sub-queries and fan them out in parallel so sub-query latency doesn't accumulate serially.
5. **Define per-task retrieval performance budgets based on the overall latency SLO:** Allocate an explicit portion of the task's latency SLO to retrieval so the iterative pattern can't silently consume the budget reserved for inference or downstream tool calls.

Resources

Related best practices:

- [AGENTPERF03-BP03 Optimize RAG retrieval pipelines for latency and precision](#)
- [AGENTPERF03-BP02 Optimize context window utilization and prompt management](#)
- [AGENTPERF02-BP01 Design efficient reasoning pipelines](#)
- [AGENTREL05-BP03 Ground agent cognition in real information](#)
- [AGENTCOST03-BP02 Cost optimize through intelligent compression and pruning of context windows](#)

Related documents:

- [Amazon Bedrock Knowledge Bases](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock Data Automation](#)

- [Agentic AI patterns and workflows on AWS](#)
- [Blog: Building intelligent search with Amazon Bedrock and Amazon OpenSearch Service for hybrid RAG solutions](#)

Related examples:

- [GitHub: Advanced RAG using Bedrock and SageMaker AI](#)

Related services:

- [Amazon Bedrock Knowledge Bases](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon OpenSearch Service](#)
- [Amazon CloudWatch](#)

Communication and protocol efficiency

Efficient communication between agents, tools, and services scales multi-agent systems from prototypes to production without latency growing alongside complexity. Agentic AI systems rely on communication between agents, tools, services, and users through various protocols and messaging patterns. The efficiency of these communication channels directly impacts agent performance. Every message exchange adds latency, every protocol handshake consumes time, and every serialization or deserialization operation uses compute resources. Optimizing communication requires selecting appropriate protocols for each interaction pattern, implementing efficient asynchronous messaging, and designing event-driven architectures that minimize polling and unnecessary processing.

AGENTPERF04: How do you achieve efficient communication and protocol usage across agent interactions?

Capability intent

- Agent-to-agent and agent-to-service communication uses asynchronous messaging by default, with synchronous calls reserved for interactions that genuinely require an immediate response.

- Protocols are selected per interaction pattern, with MCP for tool integration, A2A for agent-to-agent coordination, and streaming transports such as WebSocket for real-time user interactions.
- Event-driven integration is push-based and precisely filtered, so agents are invoked only for events that match their processing requirements and consume no compute when idle.
- Message and event payloads carry references to durable stores rather than embedded data, keeping transfer times low and letting consumers skip work on events that have become irrelevant.
- Connection pooling, protocol-level compression, and token caching keep per-interaction overhead flat as the number of agent-to-agent hops and tool invocations grows.

Maturity levels

These levels summarize what each stage of maturity looks like for communication and protocol efficiency as a whole.

Level	Name	What it looks like
1	Initial	Agent communications run over synchronous HTTP/REST with full JSON payloads on every call. Polling is the default event-detection mechanism. Connections are opened per call, and protocol choice, payload size, and authentication overhead are not tracked. Scaling problems surface only after an outage or a timeout cascade.
2	Emerging	Teams have introduced asynchronous messaging through Amazon SQS for point-to-point workflows and Amazon SNS for fan-out. Dead letter queues are attached to critical queues

Level	Name	What it looks like
		and queue depth is monitored . Most event triggers still use polling, and protocol selection is inconsistent across agents.
3	Defined	MCP is standardized for agent-to-tool communication through Amazon Bedrock AgentCore Gateway , and Amazon EventBridge with content-based filtering routes push-based events to the agents that need them. Message payloads pass references rather than data, and agents on Amazon Bedrock AgentCore Runtime use the runtime's session management for within-workflow communication. Protocol selection guidelines are documented and followed.

Level	Name	What it looks like
4	Proactive	A2A through AgentCore Runtime handles structured agent-to-agent coordination, and Amazon API Gateway WebSocket APIs serve streaming user interactions. Connection pooling, protocol-level compression, and token caching through Amazon Bedrock AgentCore Identity are standard. Idempotency keys protect against duplicate event delivery. Backpressure on queues and event-to-invocation latency are alerted on and drive automated scaling.
5	Optimized	Protocol selection, queue sizing, and event filtering are recalibrated continuously from production telemetry. Per-hop authentication and serialization overhead is budgeted and optimized at the workflow level rather than per service. Communication patterns and interoperability work feed back into internal standards and are shared with the broader agentic AI community.

Common issues to watch for

- Teams default to synchronous HTTP/REST for every agent communication, creating tight coupling where a slow downstream component blocks the entire upstream chain and propagates scaling issues across the workflow.
- Agents receive broad event streams without content-based filtering and spend compute receiving, parsing, and discarding events they never act on, which inflates cost and obscures the latency of events that do matter.
- Message and event payloads inline full data (documents, base64-encoded files, entire records) rather than passing references, so queue throughput and network transfer are consumed by payloads that consumers could fetch on demand.
- Backpressure and dead letter queues are missing, which lets fast producers overwhelm slow consumers and hides persistent message failures until a downstream timeout or data-loss incident forces investigation.
- Authentication overhead, connection setup, and serialization costs are tracked as service-level metrics rather than per-hop overheads, so a workflow with many agent-to-agent hops accumulates silent latency that no single service's telemetry surfaces.

Best practices

- [AGENTPERF04-BP01 Optimize asynchronous message handling patterns](#)
- [AGENTPERF04-BP02 Implement efficient protocol-based agent communications](#)
- [AGENTPERF04-BP03 Design high-performing event-driven integration patterns](#)

AGENTPERF04-BP01 Optimize asynchronous message handling patterns

Asynchronous messaging lets agents operate independently at their optimal pace, with message queues absorbing throughput variations and leveling load across the workflow. Synchronous request-response patterns create tight coupling that makes a slow downstream agent block the entire upstream chain. Async decouples producers from consumers so fast agents are not held up by slow ones.

Desired outcome:

- You have agent-to-agent and agent-to-service communications using asynchronous patterns by default, with synchronous communication reserved for interactions that genuinely require immediate responses.
- You have compact message payloads that pass references rather than inline data.
- You have agents processing messages at their own pace without being overwhelmed by upstream producers.

Common anti-patterns:

- Using synchronous HTTP request-response for all agent communications, creating tight coupling where a slow downstream agent blocks the entire upstream chain.
- Including large payloads (like full documents or base64-encoded files) in messages rather than passing references (like S3 URIs or document IDs) and letting the consumer retrieve the data when needed.
- Skipping backpressure mechanisms, allowing fast-producing agents to overwhelm slow-consuming agents with messages that queue up and eventually cause timeouts.

Benefits of establishing this best practice:

- Decoupled agent execution helps prevent slow agents from blocking fast agents.
- Each agent scales independently based on its own queue depth and processing rate.

Level of risk exposed if this best practice is not established: High

Implementation guidance

For agents deployed on [Amazon Bedrock AgentCore Runtime](#), the runtime's built-in session management handles message passing and state management for agent-to-agent communication inside a workflow. For workflows that need custom messaging patterns or cross-system integration, [Amazon SQS](#) provides reliable point-to-point messaging and [Amazon SNS](#) provides fan-out where a single agent event triggers multiple downstream agents. Message payloads should stay compact: pass S3 URIs, DynamoDB keys, or document IDs rather than inline data, and let the consumer retrieve the bytes on demand.

Dead letter queues (DLQs) capture messages that fail processing after retries, so failure analysis doesn't block the main flow. When consumer queues exceed depth thresholds, producers should

be throttled or consumers scaled. Amazon CloudWatch alarms on queue depth are the signal that triggers either response. For high-volume workflows, SQS batch size and long polling let you balance latency and throughput, long polling reduces empty receives, and larger batch sizes amortize request overhead across more messages.

Implementation steps

1. **Use AgentCore Runtime session management for agent-to-agent communication where possible, and use SQS or SNS for custom messaging patterns:** Let the runtime handle message passing and state inside a workflow, and fall back to Amazon SQS or Amazon SNS only for custom patterns or cross-system integration.
2. **Design compact message payloads using references rather than inline data:** Pass S3 URIs, DynamoDB keys, or document IDs in messages and let the consumer retrieve the full payload on demand.
3. **Implement dead letter queues for failed message processing with alerting on DLQ depth:** Route messages that fail after retries to a DLQ and alert when DLQ depth grows, so failure analysis happens off the main path.
4. **Add backpressure mechanisms that throttle producers when consumer queues exceed depth thresholds:** Use Amazon CloudWatch alarms on queue depth to trigger producer throttling or consumer scaling before queues reach timeout-triggering depths.
5. **Configure long polling and batch sizes for SQS consumers based on latency and throughput requirements:** Tune long polling and batch size on SQS consumers to balance empty-receive cost, per-message latency, and throughput.

Resources

Related best practices:

- [AGENTPERF04-BP02 Implement efficient protocol-based agent communications](#)
- [AGENTPERF04-BP03 Design performant event-driven integration patterns](#)
- [AGENTPERF05-BP01 Design efficient workflow orchestration patterns](#)

Related documents:

- [Building serverless architectures for agentic AI on AWS](#)
- [Foundations of agentic AI on AWS](#)

Related services:

- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon SQS](#)
- [Amazon SNS](#)
- [AWS Lambda](#)
- [Amazon CloudWatch](#)

AGENTPERF04-BP02 Implement efficient protocol-based agent communications

Standardized protocols such as the Model Context Protocol (MCP) and agent-to-agent (A2A) give agents a consistent way to communicate with tools and each other, reducing per-interaction overhead and enabling interoperability. Different protocols have different performance profiles, connection establishment, serialization, streaming, multiplexing, which makes protocol selection a meaningful performance decision for high-frequency agent communication.

Desired outcome:

- You use MCP for tool integration, A2A for agent-to-agent coordination, and streaming protocols for real-time agent-user interactions.
- You have protocol overhead minimized through connection reuse and efficient serialization.
- You have documented protocol selection guidelines for the organization.

Common anti-patterns:

- Using HTTP or REST APIs with JSON serialization for all agent communications regardless of interaction pattern, paying connection establishment and verbose serialization overhead for high-frequency internal communications.
- Implementing custom communication protocols instead of adopting standards like MCP and A2A, creating maintenance burden and blocking interoperability with the broader agent ecosystem.
- Establishing new connections for every agent interaction rather than pooling and reusing them, adding unnecessary handshake latency.

Benefits of establishing this best practice:

- Protocol-appropriate connection management reduces per-interaction overhead.
- Standard protocols open interoperability with the broader agent ecosystem.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Adopt MCP as the standard protocol for agent-to-tool communication. [Amazon Bedrock AgentCore Gateway](#) exposes tools as MCP-compatible endpoints that agents discover and invoke through a consistent interface. For agent-to-agent communication, the A2A protocol supported by AgentCore Runtime provides structured inter-agent coordination with agent card discovery, task delegation, and result collection. Frameworks such as Strands Agents and LangGraph provide MCP and A2A support.

For real-time agent-user interactions that need streaming, WebSocket connections through [Amazon API Gateway](#) keep a persistent channel open rather than reestablishing connections per turn. Connection pooling belongs on every protocol path, and protocol-level compression pays off once payloads exceed a few kilobytes.

Authentication overhead is part of the protocol performance profile. In complex multi-agent workflows, token validation, credential issuance, and policy evaluation accumulate into a measurable latency contributor. AgentCore Identity provides agent authentication with token caching, and AWS IAM roles for service-to-service authentication remove explicit credential exchange from the critical path.

Implementation steps

1. **Adopt MCP for agent-to-tool communications through AgentCore Gateway:** Expose tools as MCP-compatible endpoints through [Amazon Bedrock AgentCore Gateway](#) so agents discover and invoke tools through a consistent interface.
2. **Use A2A protocol through AgentCore Runtime for agent-to-agent coordination:** Use the A2A protocol supported by AgentCore Runtime for agent-to-agent coordination, with agent card discovery, task delegation, and result collection.
3. **Implement WebSocket connections through API Gateway for real-time streaming agent-user interactions:** Use WebSocket connections through [Amazon API Gateway](#) for real-time streaming so the channel stays open rather than reestablishing on each turn.

4. **Enable connection pooling and protocol-level compression for all communications:** Pool connections on every protocol path and enable compression once payloads exceed a few kilobytes.
5. **Budget for authentication overhead and implement token caching through AgentCore Identity:** Use AgentCore Identity with token caching for agent authentication, and use AWS IAM roles for service-to-service calls so credential exchange isn't on the critical path.

Resources

Related best practices:

- [AGENTPERF04-BP01 Optimize asynchronous message handling patterns](#)
- [AGENTPERF06-BP01 Design optimized tool integration strategies](#)
- [AGENTPERF05-BP02 Implement optimized multi-agent collaboration models](#)

Related documents:

- [Agentic AI frameworks, protocols, and tools on AWS](#)
- [Blog: Open Protocols for Agent Interoperability Part 1: Inter-Agent Communication on MCP](#)
- [Blog: Open Protocols for Agent Interoperability Part 4: Inter-Agent Communication on A2A](#)
- [Amazon Bedrock AgentCore Gateway](#)

Related videos:

- [AgentCore Deep Dive: Gateway](#)
- [Building Scalable, Self-Orchestrating AI Workflows with A2A and MCP \(DEV415\)](#)

Related examples:

- [GitHub: Amazon Bedrock AgentCore samples, Gateway tutorials](#)

Related tools:

- [Strands Agents](#)
- [LangGraph](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon API Gateway](#)

AGENTPERF04-BP03 Design high-performing event-driven integration patterns

Agents that react to events in real time, data changes, user actions, and system alerts deliver faster business outcomes than agents that poll for work. Overly broad event subscriptions trigger agents for irrelevant events, missing filtering causes agents to process and discard events they don't need. Inefficient routing adds delays that push response time past user expectations.

Desired outcome:

- You have agent workflows triggered by precisely filtered events that match their processing requirements, with minimal latency between event emission and agent invocation.
- You have efficient event schemas that include only routing metadata, with agents retrieving full context on demand.
- You have event-driven patterns that support both real-time and batch processing modes.

Common anti-patterns:

- Subscribing agents to broad event streams without filtering, forcing agents to receive and process events they immediately discard and wasting compute resources.
- Using polling-based event detection instead of push-based event delivery, adding latency and consuming compute during idle periods.
- Including full data payloads in events rather than event references, inflating event size and network transfer time when most consumers only need a subset.

Benefits of establishing this best practice:

- Push-based delivery reduces latency between event occurrence and agent invocation.
- Filtering at the event bus reduces compute waste by invoking agents only for relevant events.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Use [Amazon EventBridge](#) as the primary event bus for agent workflow triggers, with content-based filtering rules that route events to specific agents based on event attributes. For high-throughput streams, [Amazon Kinesis Data Streams](#) with Lambda event source mappings supports batch processing without forcing agents into per-event invocation. Event payloads should carry metadata, identifiers, and routing information, the full data belongs in [Amazon S3](#) or [Amazon DynamoDB](#), with references in the event. EventBridge Schema Registry standardizes event formats across the organization so consumers can generate code bindings from the schema rather than parsing one-time JSON.

For agents that need to react to database changes, [Amazon DynamoDB Streams](#) triggers agents directly from data changes without polling. Event deduplication using idempotency keys helps prevent agents from processing the same event twice, relevant whenever at-least-once delivery is the default. For complex event processing that requires correlation of multiple events before triggering an agent, [AWS Step Functions](#) or [EventBridge Pipes](#) aggregate and transform events before agent invocation.

Implementation steps

- 1. Configure EventBridge rules with content-based filtering to route events to specific agents:** Use content-based filtering on [Amazon EventBridge](#) so agents receive only the events whose attributes match their responsibilities.
- 2. Design minimal event schemas with references rather than full payloads, registered in EventBridge Schema Registry:** Keep event payloads to routing metadata and identifiers, store full data in S3 or DynamoDB, and register schemas in EventBridge Schema Registry so consumers can generate bindings.
- 3. Implement DynamoDB Streams for data-change-driven agent triggers:** Use [Amazon DynamoDB Streams](#) to trigger agents directly from data changes without polling.
- 4. Implement idempotency keys for event deduplication in agent processing logic:** Apply idempotency keys so agents don't process the same event twice under at-least-once delivery.
- 5. Monitor event processing metrics: event-to-invocation latency, filter efficiency, and throughput:** Publish these metrics to CloudWatch so filtering and routing can be tuned from measured behavior.

Resources

Related best practices:

- [AGENTPERF04-BP01 Optimize asynchronous message handling patterns](#)
- [AGENTPERF05-BP01 Design efficient workflow orchestration patterns](#)
- [AGENTPERF01-BP03 Profile end-to-end agent latency and identify optimization targets](#)

Related documents:

- [Building serverless architectures for agentic AI on AWS](#)
- [Blog: Effectively building AI agents on AWS Serverless](#)

Related services:

- [Amazon EventBridge](#)
- [Amazon Kinesis Data Streams](#)
- [AWS Lambda](#)
- [Amazon DynamoDB](#)
- [AWS Step Functions](#)

Workflow orchestration and multi-agent collaboration

The most impactful agent use cases (complex research, multi-domain customer service, and process automation) require multiple agents working together, and well-designed orchestration is what makes the whole greater than the sum of its parts. Complex agentic AI systems often involve multiple agents collaborating to solve tasks that exceed the capability of any single agent. Workflow orchestration coordinates these multi-agent interactions through patterns like supervisor-worker hierarchies, peer-to-peer collaboration, and pipeline-based task decomposition. The performance of multi-agent systems depends on how efficiently work is distributed, how agents are selected and invoked, how intermediate results are passed between agents, and how parallel execution is used to reduce latency.

AGENTPERF05: How do you optimize workflow orchestration and multi-agent collaboration for performance?

Capability intent

- Multi-agent workflows run with minimal orchestration overhead, so end-to-end latency approaches the critical path of dependent operations rather than the sum of every step.
- Orchestration patterns are matched to task shape, with dynamic graphs for reasoning-driven flows, Step Functions for deterministic skeletons, and hybrid layers for workflows that combine both.
- Independent subtasks run in parallel and large intermediate results are passed by reference, keeping the orchestration layer small and the critical path short.
- Collaboration models are matched to task characteristics (supervisor-worker, pipeline, peer-to-peer, swarm), and capabilities default to tools rather than sub-agents unless they need independent reasoning.
- Multi-stage pipelines use streaming and micro-batching to overlap stage processing, with right-sized compute per stage and end-to-end tracing that makes bottlenecks attributable.
- Delegation and handoff operations transfer only the context the receiving agent needs, through shared context stores and standardized interfaces, and handoff latency is a first-class metric.

Maturity levels

These levels summarize what each stage of maturity looks like for workflow orchestration and multi-agent collaboration as a whole.

Level	Name	What it looks like
1	Initial	Multi-agent workflows run sequentially without explicit orchestration. Teams chain agent calls in application code, pass full payloads and conversation history between steps, and use sub-agents

Level	Name	What it looks like
		for work that a tool could do in milliseconds. There is no cycle detection, no timeout, and no per-step telemetry, so runaway delegation chains and slow branches surface only as user-visible failures.
2	Emerging	Workflows are classified as dynamic, deterministic, or hybrid, and each is placed on a suitable orchestrator. AWS Step Functions is adopted for deterministic flows, and native framework orchestration is used for dynamic graphs. Basic parallelism is enabled for independent subtasks, and shared stores on Amazon S3 or Amazon DynamoDB replace inline payloads for large results. Timeouts and fallback paths exist for the most critical workflows.

Level	Name	What it looks like
3	Defined	Dynamic graph workflows run with cycle detection, maximum depth limits, and bounded fan-out cardinality. Collaboration models are selected per workflow (supervisor-worker, pipeline, peer-to-peer, swarm) using framework-native primitives such as Strands Agents agent-as-tool and Amazon Bedrock Agents multi-agent collaboration . Multi-stage pipelines use streaming through the Amazon Bedrock streaming inference API and micro-batching, with right-sized compute per stage. Shared context stores on Amazon Bedrock AgentCore Memory carry delegation context instead of inline transfers.

Level	Name	What it looks like
4	Proactive	Per-step, per-branch, and workflow-level timeouts are derived from the task SLO, and slow branches terminate with the best partial result rather than blocking. End-to-end distributed tracing through Amazon Bedrock AgentCore Observability or AWS X-Ray makes the critical path attributable, and stage rebalancing is a routine practice. Delegation happens through standardized interfaces on Amazon Bedrock AgentCore Gateway, asynchronous delegation with Amazon EventBridge callbacks is used where the parent has parallel work, and predictable receivers are pre-warmed using AWS Lambda provisioned concurrency or warm session pools.

Level	Name	What it looks like
5	Optimized	Orchestration patterns, collaboration models, and context schemas are continuously refined against measured data. Handoff latency, parallel efficiency, state payload size, and collaboration overhead metrics sit on shared dashboards that drive design iteration. New workflows start from reusable patterns, and the organization contributes reference implementations for dynamic graph orchestration, hybrid Step Functions skeletons, and standardized delegation interfaces back into the internal community.

Common issues to watch for

- Teams default to sub-agents for capabilities that a tool call would handle in milliseconds, paying full reasoning-loop cost for deterministic, single-step work.
- Workflows run steps sequentially even when the dependency graph permits parallel execution, pushing end-to-end latency to the sum of step durations rather than the critical path.
- Dynamic graph orchestrations run without cycle detection, depth limits, or bounded fan-out, so the reasoning loop occasionally produces unbounded delegation chains or excessive concurrent branches.
- Orchestrators carry large payloads inline between steps rather than passing references, inflating state size and forcing the orchestrator's context window to hold raw data it doesn't need.

- Multi-stage pipelines use identical compute configurations for every stage, over-provisioning lightweight stages and starving compute-intensive ones, and skip streaming or micro-batching that could overlap stage processing.
- Delegation transfers the full conversation history on every handoff, receiving agents re-derive context the parent already had, and handoff latency isn't measured so the overhead grows silently.

Best practices

- [AGENTPERF05-BP01 Design efficient workflow orchestration patterns](#)
- [AGENTPERF05-BP02 Implement optimized multi-agent collaboration models](#)
- [AGENTPERF05-BP03 Optimize multi-stage AI pipeline execution](#)
- [AGENTPERF05-BP04 Implement efficient agent delegation and handoff patterns](#)

AGENTPERF05-BP01 Design efficient workflow orchestration patterns

Agents that coordinate specialized sub-agents can solve complex tasks faster than any single agent, and the orchestration pattern you choose determines whether that coordination adds milliseconds or seconds. Orchestration patterns range from static workflows (the execution graph is fully defined at design time) to dynamic workflows (the agent's reasoning determines the next step at runtime). Efficient orchestration means decomposing tasks for parallelism, minimizing data passed between steps, and matching the orchestration pattern to the task's dependency structure.

Desired outcome:

- You have multi-agent workflows that execute with minimal orchestration overhead, with independent subtasks running in parallel and dependent tasks executing as soon as prerequisites complete.
- You have task routing decisions that are fast and accurate.
- You have end-to-end workflow latency that approaches the theoretical minimum defined by the critical path of dependent operations.

Common anti-patterns:

- Running all workflow steps sequentially even when some steps have no data dependencies and could run in parallel, making end-to-end latency equal to the sum of all step durations rather than the critical path.

- Using the orchestrator agent as a pass-through for all data between worker agents, creating a serialization bottleneck at every step instead of having workers write results directly to shared storage.
- Implementing dynamic graph orchestration without cycle detection or maximum depth limits, allowing LLM-driven routing decisions to produce infinite loops where agents repeatedly invoke each other without converging.

Benefits of establishing this best practice:

- Parallel execution of independent subtasks reduces end-to-end latency.
- Managed orchestration with built-in retry and error handling improves reliability.
- Scalable orchestration handles dynamic fan-out patterns without code changes.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Most agentic workflows are dynamic, meaning that the LLM's reasoning determines which tool or sub-agent to invoke next based on intermediate results. For these workflows, use your agentic framework's built-in orchestration capabilities:

- Strands Agents provides graph-based orchestration (GraphBuilder), supervisor patterns, and swarm coordination
- LangGraph offers stateful graph workflows
- CrewAI provides role-based crew orchestration

These framework-driven patterns are the natural fit for agent workloads because the execution path emerges from the model's reasoning rather than being predefined. To keep dynamic orchestration high-performing, implement cycle detection (track visited nodes with input hashing), maximum depth limits (10–15 steps), and bounded fan-out cardinality (5–10 concurrent branches) to help prevent unbounded execution chains.

For deterministic workflows where the run graph is fully known at design time, batch processing pipelines, approval workflows, data transformation chains, use [AWS Step Functions](#) with Parallel and Map states to run steps concurrently, Choice states for conditional branching, and built-in Catch and Retry for error handling. Step Functions is also valuable as a hybrid layer that handles

durable orchestration and state persistence while invoking LLM-based agents only at decision points that require reasoning.

Keep state payloads small by storing large intermediate results in [Amazon S3](#) or [Amazon DynamoDB](#) and passing only references through the orchestration layer.

For all orchestration patterns, configure per-step and workflow-level timeouts that help prevent slow agents from blocking the entire workflow. For dynamic graph workflows, allocate per-branch latency budgets derived from the overall task SLO, if a branch exceeds its budget, terminate it and proceed with the best available partial result. Design workflows to maximize parallelism by analyzing task dependencies and executing independent subtasks concurrently.

Monitor workflow execution metrics including step duration, parallel efficiency, state payload sizes, and graph depth distribution.

Implementation steps

- 1. Analyze multi-agent task dependencies and classify each workflow:** Classify each workflow as dynamic graph (LLM-driven routing, most agent workflows), hybrid (deterministic flow with LLM decision points), or static (fully predefined execution graph), and use the classification to drive the orchestration choice.
- 2. For dynamic graph workflows, use your agentic framework's native orchestration:** Use Strands GraphBuilder, LangGraph, or equivalent with cycle detection, maximum depth limits, and bounded fan-out cardinality to help prevent unbounded execution chains.
- 3. For static/hybrid workflows, implement using Step Functions with Parallel and Map states:** Use [AWS Step Functions](#) Parallel and Map states to run steps concurrently with built-in retry and error handling on deterministic paths.
- 4. Implement efficient state passing using S3/DynamoDB references rather than inline payloads:** Keep orchestration payloads small by storing large intermediate results in S3 or DynamoDB and passing only references.
- 5. Configure per-step and workflow-level timeouts with fallback strategies for slow agents:** Set timeouts at both the step and workflow level, and define fallback behavior so a single slow agent can't stall the whole workflow.
- 6. Monitor workflow execution metrics: step duration, parallel efficiency, state payload sizes, and graph depth distribution:** Publish these metrics to CloudWatch so orchestration performance can be tuned from data rather than assumption.

Resources

Related best practices:

- [AGENTPERF05-BP02 Implement optimized multi-agent collaboration models](#)
- [AGENTPERF05-BP03 Optimize multi-stage AI pipeline execution](#)
- [AGENTPERF02-BP01 Design efficient reasoning pipelines](#)

Related documents:

- [Blog: Customize agent workflows with advanced orchestration techniques using Strands Agents](#)
- [Agentic AI patterns and workflows on AWS, Workflow orchestration agents](#)
- [Building serverless architectures for agentic AI on AWS](#)
- [Blog: Effectively building AI agents on AWS Serverless](#)

Related videos:

- [Architecting scalable and secure agentic AI with AgentCore \(AIM431\)](#)
- [Build Your First Agent Workflow with Strands](#)
- [Build Reliable AI Agents with LangGraph](#)

Related examples:

- [GitHub: Guidance for multi-agent orchestration using Bedrock AgentCore](#)

Related tools:

- [Strands Agents](#)

Related services:

- [AWS Step Functions](#)
- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon S3](#)
- [Amazon DynamoDB](#)

- [Amazon CloudWatch](#)

AGENTPERF05-BP02 Implement optimized multi-agent collaboration models

Multi-agent systems typically deliver the strongest results when each collaboration pattern is matched to the task it was designed for (for example, supervisor-worker for structured decomposition, swarm for creative exploration, and pipeline for sequential processing). Before picking any multi-agent pattern, decide whether a capability should be a sub-agent or a tool that a single agent invokes. A tool call completes in milliseconds, while a sub-agent delegation is a full LLM reasoning loop that costs time and tokens.

Desired outcome:

- You have multi-agent workflows that use collaboration models matched to their task characteristics.
- You have each capability implemented at the right abstraction level, tool for deterministic single-step operations, sub-agent for tasks that require independent reasoning.
- You have coordination overhead minimized through appropriate pattern selection and implementation.

Common anti-patterns:

- Delegating to a sub-agent for capabilities that are deterministic, stateless, and single-step, API calls, database lookups, or format conversions, paying the full cost of an LLM reasoning loop for work that a tool call would handle in milliseconds.
- Using a supervisor-worker model for all multi-agent workflows, creating a bottleneck at the supervisor that must process every intermediate result and make every delegation decision.
- Deploying swarm patterns without explicit convergence criteria or resource budgets, letting agents continue exploring indefinitely without converging on a shared outcome.

Benefits of establishing this best practice:

- Matching the collaboration model to task structure minimizes coordination overhead.
- Appropriate model selection for decomposable tasks maximizes parallelism.

- Timeouts and fallback mechanisms keep collaboration resilient when individual agents fail or slow down.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Use a sub-agent when the capability requires its own reasoning (LLM inference for ambiguous inputs or judgment calls), its own context or memory scope, multi-step tool orchestration where sequencing requires reasoning, a different model or prompt, or independent failure isolation.

Use a tool when the capability is deterministic, stateless, single-step, and fast (under 2-3 seconds).

For borderline cases, start with a tool and promote to a sub-agent only when frequent re-invocation patterns indicate the capability needs its own reasoning loop.

Once multi-agent architecture is warranted, select the collaboration model based on task characteristics. Use supervisor-worker when the task has clear decomposition and centralized quality control is needed. Strands Agents's agent-as-tool pattern and Amazon Bedrock Agents' multi-agent collaboration provide supervisor-worker orchestration with automatic context passing and result aggregation.

Use pipeline when the task has a natural sequential flow, balance stage durations and use your framework's graph orchestration to chain agents sequentially.

Use peer-to-peer or blackboard when multiple agents need to contribute partial solutions asynchronously, implement a shared workspace using AgentCore Memory or DynamoDB with event-driven notifications.

Use swarm when the task benefits from parallel exploration and emergent behavior, implement convergence detection and per-swarm token budgets to help prevent unbounded resource consumption.

For deterministic delegation logic or durable long-running workflows, [AWS Step Functions](#) remains a strong alternative for orchestrating agent invocations.

For all collaboration models, implement timeout and fallback mechanisms that help prevent a single slow or failed agent from blocking the entire workflow. Deploy multi-agent systems on [Amazon Bedrock AgentCore Runtime](#) for managed scaling and observability, or on Amazon EKS or Amazon ECS for custom container-based deployments. Monitor collaboration overhead metrics including coordination latency, redundant work rate, and throughput per model.

Implementation steps

1. **Evaluate whether each capability should be a tool or a sub-agent:** Default to tools and promote to sub-agents only when re-invocation patterns indicate the need for independent reasoning.
2. **Classify multi-agent workflows by task characteristics:** Assess decomposability, dependency structure, latency requirements, and whether the task benefits from parallel exploration.
3. **Select the collaboration model:** Use supervisor-worker for clear decomposition, pipeline for sequential flow, peer-to-peer for shared problem spaces, and swarm for parallel exploration.
4. **Implement using your framework's native multi-agent patterns:** Use Strands agent-as-tool, Amazon Bedrock multi-agent collaboration, or an equivalent, and use [Amazon Bedrock AgentCore Memory](#) for shared context across agents.
5. **Implement timeout and fallback mechanisms for all collaboration models:** Set per-agent timeouts and define fallback behavior so one slow or failed agent can't block the full workflow.
6. **Monitor collaboration overhead metrics and optimize model selection over time:** Track coordination latency, redundant work rate, and throughput per model, and re-evaluate the collaboration choice as traffic shifts.

Resources

Related best practices:

- [AGENTPERF05-BP01 Design efficient workflow orchestration patterns](#)
- [AGENTPERF05-BP04 Implement efficient agent delegation and handoff patterns](#)
- [AGENTPERF04-BP02 Implement efficient protocol-based agent communications](#)

Related documents:

- [Blog: Multi-agent collaboration patterns with Strands Agents and Amazon Nova](#)
- [Blog: Multi-agent collaboration with Strands](#)
- [Agentic AI patterns and workflows on AWS, Multi-agent collaboration](#)
- [Agentic AI patterns and workflows on AWS, Workflow orchestration agents](#)
- [Use multi-agent collaboration with Amazon Bedrock Agents](#)

Related videos:

- [Amazon Bedrock Agents and AgentCore Design Patterns \(TNC322\)](#)

Related examples:

- [GitHub: Bedrock multi-agent collaboration workshop](#)
- [GitHub: Multi-agent collaboration with Strands](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock Agents](#)
- [Amazon Bedrock AgentCore](#)
- [AWS Step Functions](#)
- [Amazon DynamoDB](#)

AGENTPERF05-BP03 Optimize multi-stage AI pipeline execution

Real-world agent tasks rarely complete in a single step. Document processing, data analysis, and customer service workflows all involve multiple sequential stages where each stage's throughput is limited by the slowest process or mechanism. Each stage transition introduces overhead (like serialization, network transfer, or cold starts), and streaming or micro-batching allows downstream stages to begin processing before upstream stages complete, overlapping execution to cut total latency.

Desired outcome:

- You have multi-stage AI pipelines that execute with minimal inter-stage overhead, with data flowing efficiently between stages.
- You have pipeline throughput balanced across stages with no single stage creating a persistent bottleneck.
- You have streaming implemented where possible to overlap processing.
- You have each stage's compute resources right-sized for its specific requirements.

Common anti-patterns:

- Waiting for an entire batch to complete one stage before starting the next, when streaming or micro-batching would let downstream stages begin processing as upstream results become available.
- Using the same compute configuration for all pipeline stages regardless of their processing requirements, over-provisioning lightweight stages and under-provisioning compute-intensive stages.
- Serializing large intermediate results to persistent storage between every stage when in-memory passing or streaming would be more efficient for stages that execute in close succession.

Benefits of establishing this best practice:

- Streaming and micro-batching overlap stage processing, reducing end-to-end latency.
- Balanced stage capacity and buffered inter-stage communication improve throughput.
- Right-sized compute per stage optimizes cost.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Implement multi-stage pipelines using [AWS Step Functions](#) with stage-specific [AWS Lambda](#) functions, [Amazon ECS](#) tasks, or agents hosted on [Amazon Bedrock AgentCore Runtime](#), where each stage's compute configuration is independently tuned. For pipelines with high throughput requirements, Step Functions Distributed Map processes items in parallel across stages. Right-size compute for each stage, using Lambda for lightweight processing, ECS for compute-intensive stages, and AgentCore Runtime for stages that require LLM-based reasoning.

Streaming between stages is the single largest latency win when it applies. Use Amazon Bedrock's streaming inference API to begin post-processing output tokens as they are generated rather than waiting for the complete response. For data-intensive pipelines, [Amazon Kinesis Data Streams](#) acts as an inter-stage buffer that supports streaming data flow, so downstream stages begin processing as soon as upstream results are available. For batch pipelines, micro-batching sends small groups of items to downstream stages as they complete rather than waiting for the entire batch.

Pipeline-level observability through [Amazon Bedrock AgentCore Observability](#) or [AWS X-Ray](#) traces requests across all stages, identifying the critical path and the stage that contributes most

to end-to-end latency. Balance stage durations by profiling each stage and adjusting processing granularity, split slow stages into parallel sub-stages or combine fast stages to reduce inter-stage transitions.

Implementation steps

1. **Map the multi-stage pipeline and identify dependencies between stages:** Document stage dependencies, opportunities for streaming, and the critical path so optimization effort lands on the stages that drive end-to-end latency.
2. **Implement each stage as an independent compute unit with stage-specific resource configurations:** Use [AWS Lambda](#) for lightweight processing, [Amazon ECS](#) for compute-intensive stages, and AgentCore Runtime for stages that require LLM reasoning, and tune each stage's resources to its own profile.
3. **Enable streaming between stages using the Amazon Bedrock streaming API and Kinesis Data Streams where applicable:** Use the streaming inference API to post-process output tokens as they are generated, and [Amazon Kinesis Data Streams](#) as an inter-stage buffer so downstream stages begin processing as upstream results arrive.
4. **Implement micro-batching for batch pipelines to reduce end-to-end latency:** Send small groups of items to downstream stages as they complete rather than waiting for the full batch.
5. **Configure AgentCore Observability or X-Ray tracing across all pipeline stages for end-to-end latency visibility:** Use [Amazon Bedrock AgentCore Observability](#) or [AWS X-Ray](#) to trace requests across every stage.
6. **Monitor per-stage latency, throughput, and resource utilization to identify and resolve bottlenecks:** Publish metrics for each stage so bottleneck stages are visible and can be split, parallelized, or resized.

Resources

Related best practices:

- [AGENTPERF05-BP01 Design efficient workflow orchestration patterns](#)
- [AGENTPERF02-BP04 Optimize streaming responses and time-to-first-token for agent interactions](#)
- [AGENTPERF02-BP03 Optimize agent execution paths for reduced latency](#)

Related documents:

- [Multi-stage AI workflow pattern, Building serverless architectures for agentic AI](#)
- [Blog: Effectively building AI agents on AWS Serverless](#)
- [Agentic AI patterns and workflows on AWS](#)

Related examples:

- [GitHub: Build GenAI agent workflows with Step Functions](#)

Related services:

- [AWS Step Functions](#)
- [AWS Lambda](#)
- [Amazon ECS](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon Kinesis Data Streams](#)
- [Amazon Bedrock](#)

AGENTPERF05-BP04 Implement efficient agent delegation and handoff patterns

Smooth agent-to-agent transitions make multi-agent workflows feel like a single cohesive experience, where the receiving agent picks up exactly where the delegating agent left off. Delegation and handoff both require efficient context transfer. The receiving agent needs enough context to act, but transferring too much wastes time and tokens.

Desired outcome:

- You have agent delegation and handoff operations that complete with minimal latency, transferring precisely the context needed by the receiving agent.
- You have receiving agents that begin productive processing immediately without re-deriving context the delegating agent already possessed.
- You have standardized context transfer mechanisms that let any agent delegate to or receive handoffs from any other agent.
- You have handoff latency measured and optimized as part of the overall workflow performance budget.

Common anti-patterns:

- Transferring the entire conversation history and all accumulated context during every delegation, regardless of what the receiving agent actually needs, wasting serialization time and context window capacity.
- Requiring receiving agents to re-derive context (re-query databases, re-retrieve documents) that the delegating agent already had, duplicating work and adding latency.
- Implementing delegation as synchronous blocking calls where the parent agent waits idle for the child agent to complete, wasting the parent's compute resources.

Benefits of establishing this best practice:

- Selective context transfer and shared context stores reduce delegation latency.
- Receiving agents reuse context already gathered by delegating agents instead of repeating the work.
- Asynchronous delegation patterns improve parent agent throughput.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Implement a shared context store using [Amazon Bedrock AgentCore Memory](#) or [Amazon DynamoDB](#) where delegating agents write context and receiving agents read it, avoiding the need to serialize and transfer large context payloads through the orchestration layer. Context transfer schemas define the minimum context required for each delegation type, a data validation agent needs the data and validation rules, not the full conversation history. For agents built with Strands Agents, the built-in agent-as-tool pattern automatically inherits relevant context from the parent agent's session.

For handoff patterns in conversational agents, context summarization compresses the conversation into a concise handoff summary tailored to the receiving agent's role, rather than transferring raw conversation history. For predictable delegation patterns, for example, a triage agent that consistently delegates to one of several specialist agents, pre-warming through [AWS Lambda](#) provisioned concurrency or warm session pools on AgentCore Runtime removes cold-start latency from the receiving side. Asynchronous delegation lets the parent agent continue processing other tasks while the child agent works, using callbacks or [Amazon EventBridge](#) notifications to receive results.

[Amazon Bedrock AgentCore Gateway](#) standardizes delegation interfaces, letting any agent delegate to any other agent through a consistent API that handles context transfer, authentication, and result delivery. Handoff latency belongs in agent performance dashboards as a distinct metric, measured from delegation initiation to the receiving agent's first productive action.

Implementation steps

1. **Identify delegation and handoff patterns in existing multi-agent workflows and measure current transition latency:** Map delegation and handoff points and measure the current transition latency so optimization targets are grounded in data.
2. **Implement shared context stores using AgentCore Memory or DynamoDB for context transfer between agents:** Use [Amazon Bedrock AgentCore Memory](#) or [Amazon DynamoDB](#) so delegating agents write context once and receiving agents read it without serializing large payloads.
3. **Define minimum context schemas for each delegation type, specifying exactly what the receiving agent needs:** Keep the delegation payload small and purpose-specific so receivers only get the context required for their role.
4. **Implement context summarization for conversational handoffs that compresses history into role-appropriate summaries:** Summarize raw conversation history into a handoff summary tailored to the receiving agent's role rather than forwarding the full transcript.
5. **Configure pre-warming for predictable delegation patterns using provisioned concurrency or warm session pools:** For recurring delegation targets, use [AWS Lambda](#) provisioned concurrency or warm session pools on AgentCore Runtime to remove cold-start latency.
6. **Convert synchronous delegations to asynchronous patterns with callback-based result delivery:** Let the parent agent continue other work while the child agent runs, receiving results through callbacks or EventBridge notifications.

Resources

Related best practices:

- [AGENTPERF05-BP01 Design efficient workflow orchestration patterns](#)
- [AGENTPERF05-BP02 Implement optimized multi-agent collaboration models](#)
- [AGENTPERF04-BP01 Optimize asynchronous message handling patterns](#)

Related documents:

- [Blog: Multi-agent collaboration patterns with Strands Agents and Amazon Nova](#)
- [Agentic AI patterns and workflows on AWS, Multi-agent collaboration](#)
- [Operationalizing agentic AI on AWS, Design for composability and collaboration](#)
- [Amazon Bedrock AgentCore Memory](#)

Related videos:

- [AgentCore Memory: Episodic Memory & Patterns](#)

Related examples:

- [GitHub: Guidance for multi-agent orchestration using Bedrock AgentCore](#)
- [GitHub: Amazon Bedrock AgentCore samples, Memory tutorials](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon DynamoDB](#)
- [AWS Lambda](#)

Tool integration and framework optimization

Agents extend their capabilities by invoking external tools (APIs, databases, search engines, code interpreters, and other services) and operate within frameworks that provide the scaffolding for agent behavior. The performance of tool integrations directly impacts agent responsiveness because tool invocations occur within the agent reasoning loop, adding latency to every iteration that requires external capabilities. Framework selection and configuration similarly affect performance through their impact on agent loop efficiency, memory management, and orchestration overhead. Optimizing this layer requires selecting frameworks that minimize overhead, designing tool integration strategies that reduce invocation latency, implementing

efficient tool discovery and selection mechanisms, and using meta-tools that compose multiple capabilities into single efficient operations.

AGENTPERF06: How do you optimize tool integrations and framework usage for agent performance?

Capability intent

- Tool invocations add minimal latency to the agent reasoning loop, and the selection experience is consistent across agents regardless of where a tool is implemented.
- Agents see a filtered catalog of 5 to 10 relevant tools per task context rather than the full catalog, keeping tool selection accuracy high as the catalog grows.
- Independent tool calls run in parallel, and reusable results are cached at the scope that matches the data's semantics (request, session, or global).
- Tool invocation patterns (connection pooling, timeouts sized to real behavior, backoff and jitter on retries, automatic cutoffs) keep infrastructure overhead off the critical path.
- Meta-tools encapsulate predictable multi-step sequences into single server-side operations, collapsing reasoning iterations for routine work while individual tools remain available for novel cases.
- Per-tool latency, error rate, cutoff state, and cache effectiveness are instrumented, so tool-layer regressions are attributable before they dominate task latency.

Maturity levels

These levels summarize what each stage of maturity looks like for tool integration and framework optimization as a whole.

Level	Name	What it looks like
1	Initial	Tools are integrated directly by each agent team, with no shared protocol and no catalog discipline. The full tool list is presented to the model on every reasoning

Level	Name	What it looks like
		iteration, tool calls run sequentially, and results are recomputed on every invocation. Connections are re-established per call, timeouts default to SDK values that are too generous, and retries run without backoff. There is no per-tool telemetry, so regressions surface only as user-facing slowness.
2	Emerging	Model Context Protocol (MCP) is adopted as the integration protocol and Amazon Bedrock AgentCore Gateway exposes tools through a consistent interface. Semantic tool discovery is used for large catalogs so the model sees a filtered set rather than every tool. Basic connection pooling is in place for custom endpoints, and per-tool timeouts are set from measured p95 latency. Retries use backoff and jitter, and per-tool latency and error rate metrics are published through Amazon CloudWatch .

Level	Name	What it looks like
3	Defined	Parallel tool execution is enabled for independent calls. Tool APIs are designed for agent consumption with compact schemas, pagination, and partial responses. Results are cached at multiple scopes (request, session, global) using in-process caches and Amazon ElastiCache where appropriate. Batch APIs replace single-item loops for multi-item work, and AWS Lambda -based tools initialize clients outside the handler to preserve connection pools across invocations.

Level	Name	What it looks like
4	Proactive	Automatic cutoffs route around degraded tools, falling back to cached results or controlled errors rather than exhausting the reasoning budget. Frequently repeated tool sequences are identified from telemetry and consolidated into meta-tools deployed on AWS Lambda and exposed through Amazon Bedrock AgentCore Gateway. Meta-tool performance is compared against the equivalent individual-tool sequence, and per-tool dashboards alarm on latency, error rate, timeout rate, and cutoff state regressions.

Level	Name	What it looks like
5	Optimized	Tool integration, invocation, and meta-tool strategies are continuously refined against operational data. Meta-tools are retired when they stop outperforming the individual-tool baseline, cache TTLs are tuned per tool, and tool catalogs evolve with usage patterns. The organization publishes reusable agent-friendly tool patterns (compact schemas, cutoff wrappers, meta-tool templates) and shares benchmarks across teams, keeping the tool-layer contribution to end-to-end latency at the lower bound defined by each tool's inherent processing time.

Common issues to watch for

- Teams present every available tool to the model on every reasoning iteration, consuming context window capacity and degrading selection accuracy once the catalog grows beyond 10 to 15 tools.
- Independent tool calls run sequentially rather than in parallel, so tool-layer latency compounds linearly with the number of calls instead of tracking the maximum.
- Tool result caching is skipped, so agents re-invoke the same tool with identical parameters several times within a single task.
- Connections are re-established on every invocation, adding TLS handshake latency to every tool call that could have reused a persistent connection.
- Timeouts are defaulted or set too generously, letting a single pathological tool call consume the entire task latency budget, and retries run without backoff or a total budget.

- Predictable multi-step tool sequences are re-executed as individual tool calls on every occurrence instead of being consolidated into meta-tools, wasting reasoning iterations and tokens on routine work.

Best practices

- [AGENTPERF06-BP01 Design optimized tool integration strategies](#)
- [AGENTPERF06-BP02 Implement efficient tool invocation patterns](#)
- [AGENTPERF06-BP03 Optimize meta-tool utilization and tool chaining](#)

AGENTPERF06-BP01 Design optimized tool integration strategies

Agents that surface the right tools at the right time respond faster and make better decisions. LLMs make increasingly poor tool selection decisions once the candidate set grows beyond 10-15 tools, so dynamic filtering, parallel execution, and cached results keep the reasoning loop tight even as the tool catalog grows. Tool invocations happen inside the reasoning loop, which means their latency adds directly to response time.

Desired outcome:

- You have tool invocations that add minimal latency to the agent reasoning loop.
- You have tool selection that is fast and accurate, with agents choosing from a filtered set of 5-10 relevant options rather than evaluating the full catalog.
- You have independent tool calls executing in parallel.
- You have tool results cached where appropriate.
- You have per-tool latency, error rate, and usage metrics providing visibility into tool performance.

Common anti-patterns:

- Presenting all available tools to the agent on every reasoning iteration, forcing the LLM to evaluate dozens of tool descriptions, consuming context window capacity and degrading selection accuracy.
- Executing tool calls sequentially when they have no data dependencies, adding latency equal to the sum of all tool durations rather than the maximum.

- Skipping tool result caching, so agents re-invoke the same tool with identical parameters multiple times within a single task.

Benefits of establishing this best practice:

- Parallel tool execution and result caching reduce reasoning loop latency.
- Dynamic filtering that presents only relevant tools improves tool selection accuracy.
- Per-tool monitoring speeds detection of tool performance issues.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Adopt MCP as the standard tool integration protocol and use [Amazon Bedrock AgentCore Gateway](#) to expose tools as MCP-compatible endpoints. AgentCore Gateway provides built-in semantic tool discovery (`x_amz_bedrock_agentcore_search`) so agents query for relevant tools by natural language description rather than receiving the full catalog. For agents with access to large tool catalogs, a two-stage selection pattern works well: a lightweight pre-filter narrows the full catalog to the 5-10 most relevant tools based on current task context, and only those filtered tools appear in the LLM's prompt. For agents built with Strands Agents or another agentic framework, built-in parallel tool execution runs independent tool calls concurrently.

Design tool APIs specifically for agent consumption, like compact response schemas that return only the fields the agent needs, pagination for large result sets, and partial response support. Cache tool results at multiple levels, request-scoped (within a single reasoning session), session-scoped (across reasoning iterations for the same user), and global (shared data with appropriate TTLs). Tool health monitoring tracks per-tool latency, error rates, and availability, and automatic cutoffs route around slow or failing tools.

Implementation steps

1. **Adopt MCP as the standard tool integration protocol and expose tools through AgentCore Gateway:** Expose tools as MCP-compatible endpoints through [Amazon Bedrock AgentCore Gateway](#) so agents use a single consistent interface.
2. **Enable AgentCore Gateway's semantic tool discovery to filter tools by task relevance:** Use `x_amz_bedrock_agentcore_search` to narrow the tool set per request so the LLM evaluates only the most relevant 5-10 tools.

3. **Implement parallel tool execution for independent tool calls within the same reasoning step:** Use your framework's native parallel tool execution (Strands Agents, LangGraph) so independent tool calls run concurrently.
4. **Deploy multi-level tool result caching with appropriate TTLs per tool:** Cache tool results at request-, session-, and global-scope with TTLs matched to each tool's freshness requirements.
5. **Configure tool health monitoring with per-tool latency and error rate metrics, and automatic cutoffs for degraded tools:** Track per-tool latency and error rate in Amazon CloudWatch and use automatic cutoffs to route around degraded tools.

Resources

Related best practices:

- [AGENTPERF06-BP02 Implement efficient tool invocation patterns](#)
- [AGENTPERF06-BP03 Optimize meta-tool utilization and tool chaining](#)
- [AGENTPERF04-BP02 Implement efficient protocol-based agent communications](#)

Related documents:

- [Blog: Introducing Amazon Bedrock AgentCore Gateway: Transforming enterprise AI agent tool development](#)
- [Blog: Transform your MCP architecture: Unite MCP servers through AgentCore Gateway](#)
- [Blog: Open Protocols for Agent Interoperability Part 3: Strands Agents & MCP](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock Data Automation](#)
- [Agentic AI frameworks, protocols, and tools on AWS](#)

Related videos:

- [Scale agent tools with AgentCore Gateway \(AIM3313\)](#)
- [Integrating MCP Tools with Strands Agents](#)
- [Strands Tools: Building Custom AI Agents with Python](#)

Related examples:

- [GitHub: Amazon Bedrock AgentCore samples, Gateway tutorials](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore, Lab 3: Gateway, Identity & Policy](#)
- [Diving Deep into Bedrock AgentCore, Gateway](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon ElastiCache](#)
- [Amazon CloudWatch](#)

AGENTPERF06-BP02 Implement efficient tool invocation patterns

Well-tuned tool invocation patterns help the agent's responsiveness reflect the tool's actual processing time, not infrastructure overhead. Each tool invocation involves connection establishment, serialization, network transfer, processing, and deserialization, trimming each component compounds across the many tool calls in a typical agent task.

Desired outcome:

- You have individual tool invocations that execute with minimal overhead beyond the tool's inherent processing time.
- You have connection pooling that removes repeated connection establishment costs.
- You have timeouts that help prevent slow tools from blocking agent execution.
- You have automatic cutoffs that detect degraded tools and route to alternatives.
- You have per-tool invocation metrics that provide visibility into performance characteristics.

Common anti-patterns:

- Establishing new connections for every tool invocation rather than maintaining connection pools, adding hundreds of milliseconds of TLS handshake latency to each call.
- Implementing aggressive retry strategies without backoff or jitter, creating retry storms that overwhelm already-degraded tools.
- Setting tool invocation timeouts too high or not at all, letting a single slow tool call block the agent for seconds and exceed the overall task latency budget.

Benefits of establishing this best practice:

- Connection pooling and persistent connections reduce per-tool-call overhead.
- Appropriate timeouts protect the agent latency budget.
- Automatic cutoffs support fast failover to alternatives.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

For tools accessed through [Amazon Bedrock AgentCore Gateway](#), the gateway handles connection management, authentication, and routing automatically. For custom tool endpoints, connection pooling through HTTP keep-alive connections or framework-specific pool configurations keeps TLS sessions warm across invocations.

For [AWS Lambda](#)-based tools, initializing connections outside the handler function so they persist across invocations within the same execution environment turns a handshake for each request into one per environment. Tool invocation timeouts should be set based on the tool's expected response time, typically two to three times the p95 latency.

Retry strategies with exponential backoff and jitter handle transient failures, with a maximum of two to three retries and a total retry budget that doesn't exceed the tool's timeout. Automatic cutoff patterns track tool failure rates and open the circuit when failures exceed a threshold, returning a cached result or error immediately rather than waiting for another timeout.

For tools that support batch operations, request batching (a batch API for multiple item lookups) reduces overhead for each call across the set. Per-tool invocation metrics (latency percentiles, error rates, timeout rates, and automatic cutoff state) belong on the agent performance dashboard alongside reasoning-loop metrics.

Implementation steps

1. **Use AgentCore Gateway for managed tool access where possible, and implement connection pooling for custom tool endpoints:** Use [Amazon Bedrock AgentCore Gateway](#) for managed tool access. For custom endpoints, enable HTTP keep-alive and framework-level connection pools.
2. **Configure per-tool timeouts based on profiled p95 latency:** Size each tool's timeout to two to three times its measured p95 latency so slow individual calls don't stall the agent.
3. **Implement retry strategies with exponential backoff and jitter:** Use exponential backoff with jitter and cap retries at two to three so transient failures recover without overwhelming already-degraded tools.
4. **Deploy automatic cutoff patterns that fast-fail when tools are degraded:** Track failure rate per tool and open the circuit when failures exceed a threshold, returning a cached result or error immediately.
5. **Implement request batching for tools that support batch operations:** Use batch APIs when multiple items are needed so per-call overhead amortizes across the set.
6. **Monitor per-tool invocation metrics and establish alerting for latency and error rate anomalies:** Publish per-tool latency percentiles, error rates, timeout rates, and cutoff state to CloudWatch with alarms on anomalies.

Resources

Related best practices:

- [AGENTPERF06-BP01 Design optimized tool integration strategies](#)
- [AGENTPERF02-BP03 Optimize agent execution paths for reduced latency](#)
- [AGENTPERF03-BP04 Establish efficient agent caching and data access patterns](#)

Related documents:

- [Blog: Build long-running MCP servers on Amazon Bedrock AgentCore with Strands Agents integration](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Agentic AI frameworks, protocols, and tools on AWS](#)

Related examples:

- [GitHub: Amazon Bedrock AgentCore samples, Gateway tutorials](#)

Related tools:

- [Strands Agents](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Lambda](#)
- [Amazon CloudWatch](#)

AGENTPERF06-BP03 Optimize meta-tool utilization and tool chaining

Meta-tools let agents accomplish in one reasoning step what would otherwise take five. For tasks that require a predictable sequence of tool calls, a meta-tool combines the entire sequence into a single server-side operation and returns the final result in one reasoning iteration instead of many, cutting both latency and token cost by removing intermediate reasoning steps.

Desired outcome:

- You have common multi-step tool sequences encapsulated as meta-tools that execute the full sequence in a single agent reasoning iteration.
- You have agents using meta-tools for routine operations and individual tools for novel or unpredictable tasks.
- You have meta-tool performance monitored to validate that the composite operation is faster than the equivalent sequence of individual tool calls.

Common anti-patterns:

- Requiring the agent to make individual tool calls for every step of a predictable sequence (for example, search, retrieve, parse, and format), consuming multiple reasoning iterations for what could be a single meta-tool invocation.
- Creating overly complex meta-tools that try to handle too many variations, becoming hard to maintain and slower than individual tools for edge cases.

- Skipping meta-tools for frequently repeated tool sequences, forcing agents to re-discover and re-execute the same tool chain on every occurrence.

Benefits of establishing this best practice:

- Meta-tools that collapse multi-step sequences into single invocations reduce LLM inference calls.
- Removing intermediate reasoning iterations lowers latency and token cost.

Level of risk exposed if this best practice is not established: Low

Implementation guidance

Analyze agent telemetry to identify frequently repeated tool call sequences, patterns where agents consistently call the same tools in the same order with predictable data flow between them.

Implement these sequences as meta-tools using [AWS Lambda](#) functions that execute the entire sequence server-side and return the final result.

For example, a "research" meta-tool might combine knowledge base search, document retrieval, and relevance extraction into a single invocation.

Expose meta-tools through MCP through [Amazon Bedrock AgentCore Gateway](#) alongside individual tools, so agents can choose between the meta-tool for routine operations and individual tools for novel tasks that require step-by-step reasoning.

Design meta-tools with clear input and output contracts and error handling that provides meaningful feedback when any step in the sequence fails. Update agent prompts to include meta-tool descriptions that guide the agent to prefer them for routine operations.

Monitor meta-tool performance to validate that the composite operation is faster than the equivalent individual tool sequence, and decompose meta-tool latency into per-step metrics so optimization is directed at the slowest step.

Implementation steps

1. **Analyze agent telemetry to identify frequently repeated tool call sequences:** Look for sequences that occur three or more times with predictable data flow between steps.
2. **Design meta-tools for the most common sequences with clear input/output contracts and error handling:** Define explicit contracts and per-step error handling so meta-tool failures are attributable.

3. **Implement meta-tools as Lambda functions that execute the full sequence server-side:** Use [AWS Lambda](#) to run the sequence server-side so the agent sees one call instead of many.
4. **Expose meta-tools through MCP and AgentCore Gateway alongside individual tools:** Register meta-tools with [Amazon Bedrock AgentCore Gateway](#) alongside individual tools so the agent can choose based on task.
5. **Monitor meta-tool performance and compare against equivalent individual tool sequences:** Track meta-tool latency and per-step breakdown and compare against the individual-tool path to confirm the meta-tool is actually faster.

Resources

Related best practices:

- [AGENTPERF06-BP01 Design optimized tool integration strategies](#)
- [AGENTPERF06-BP02 Implement efficient tool invocation patterns](#)
- [AGENTPERF05-BP03 Optimize multi-stage AI pipeline execution](#)

Related documents:

- [Blog: Flexibility to Framework: Building MCP Servers with Controlled Tool Orchestration](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Agentic AI frameworks, protocols, and tools on AWS](#)

Related videos:

- [AgentCore Deep Dive: Browser Tool & Code Interpreter](#)
- [Excel-lent Agents: AgentCore's Code Interpreter](#)

Related examples:

- [GitHub: Amazon Bedrock AgentCore samples, Gateway tutorials](#)

Related tools:

- [Strands Agents](#)

Related services:

- [AWS Lambda](#)
- [Amazon Bedrock AgentCore](#)

Multi-tenancy and resource optimization

A well-designed multitenant agent service lets organizations serve many teams and customers from shared infrastructure while delivering consistent, predictable performance to each. Multitenant agent deployments must isolate one tenant's workload so it can't degrade the performance experienced by other tenants, and at the same time optimize overall resource utilization to control costs. This requires designing deployment models that provide appropriate isolation boundaries, implementing throttling and quota mechanisms that enforce fair resource allocation, and right-sizing serverless compute configurations for the bursty nature of agent workloads.

AGENTPERF07: How do you manage multitenant performance isolation and optimize resource utilization?

Capability intent

- Tenant deployments are organized into tiers with explicit performance, isolation, and cost profiles, rather than a single shape applied uniformly to every tenant.
- Pooled tiers share managed infrastructure that provides session-level isolation, and premium or enterprise tiers add dedicated resources where stronger isolation is required.
- Throttling is enforced at every shared resource layer (API, inference, memory, tools), so tenants can't bypass edge limits through long-running downstream operations.
- Adaptive throttling lets tenants use slack capacity during low-load periods and snaps back to baseline during contention, while honoring tier priority.
- Tenant context propagates through the full stack, and per-tenant dashboards track latency, error rates, throttle state, and resource consumption against the tier's SLA.
- Isolation is validated through regular noisy-neighbor testing, so the deployment model's performance claims are measured rather than assumed.

Maturity levels

These levels summarize what each stage of maturity looks like for multi-tenancy and resource optimization as a whole.

Level	Name	What it looks like
1	Initial	Tenants are served from a single shared deployment with no isolation mechanisms or from duplicated siloed infrastructure with no tier policy. Throttling, if it exists, is at the API edge only and uses static limits. There is no per-tenant telemetry, so noisy-neighbor effects and SLA violations are discovered after customers report them. Tenant onboarding requires manual infrastructure provisioning for every new customer.
2	Emerging	A small number of tiers (standard and premium, optionally enterprise) is documented with performance SLAs, isolation requirements, and pricing. The standard tier runs on pooled infrastructure such as Amazon Bedrock AgentCore Runtime with session-level isolation. Amazon API Gateway usage plans enforce per-tenant rate limits at the edge. Data

Level	Name	What it looks like
		is isolated per tier using Amazon DynamoDB partition keys for pooled tiers.
3	Defined	Throttling is enforced at every shared layer, not only at the edge. Tenant-aware queuing limits per-tenant concurrency on Amazon Bedrock invocations, and memory and tool layers enforce their own per-tenant limits. Premium tenants use dedicated resources such as Amazon Bedrock provisioned throughput . Tenant context propagates through the runtime, memory, and tool layers. Per-tenant dashboards in Amazon CloudWatch track consumption and SLA compliance.

Level	Name	What it looks like
4	Proactive	Adaptive throttling loosens limits during low-load periods and tightens during contention, honoring tier priority so premium tenants retain capacity. Onboarding for standard-tier tenants is configuration-driven through AWS CDK or AWS CloudFormation templates rather than a bespoke infrastructure build. Alarms fire on per-tenant limit proximity and SLA violations, and noisy-neighbor testing runs on a regular cadence to validate isolation claims against real load.

Level	Name	What it looks like
5	Optimized	Tier definitions, throttling policies, and deployment models are continuously refined against operational data. Onboarding is fully self-service for standard and premium tiers with guardrails that help prevent tier-inappropriate configurations. Cost and performance per tenant are reported to tenant-facing dashboards, and isolation effectiveness is validated through automated chaos and noisy-neighbor scenarios. The organization publishes reusable multitenant reference patterns internally and contributes benchmarks across teams.

Common issues to watch for

- Teams deploy fully isolated infrastructure for every tenant regardless of need, wasting capacity on tenants whose requirements would be met by pooled resources.
- Teams use a single pooled deployment without per-layer isolation, letting high-volume tenants consume disproportionate capacity and degrade performance for everyone else.
- Throttling is enforced only at the API edge, so tenants bypass edge limits through long-running inference, memory, or tool operations that saturate downstream shared resources.
- Throttling uses static limits that neither use available slack during low-load periods nor protect isolation during high-load periods, and premium tenants are throttled identically to standard tenants.
- Tenant onboarding requires infrastructure provisioning on every new customer, so onboarding time and operational overhead grow linearly with tenant count.

- Isolation is assumed rather than tested, so subtle changes (a new tool, a heavier prompt, a shift in traffic mix) erode the isolation boundary silently and surface only during a real incident.

Best practices

- [AGENTPERF07-BP01 Design efficient multitenant agent deployment models](#)
- [AGENTPERF07-BP02 Implement tenant-aware performance isolation and throttling](#)

AGENTPERF07-BP01 Design efficient multitenant agent deployment models

Organizations that serve multiple tenants from a shared agent service get better resource efficiency and faster tenant onboarding when the deployment model delivers consistent performance for every tenant. Siloed deployments provide strong isolation at higher cost. Pooled deployments maximize efficiency but need mechanisms to help prevent noisy neighbor effects. Hybrid models combine both, using pooled resources for standard tenants and dedicated resources for premium tenants.

Desired outcome:

- You have multitenant agent deployments that use deployment models matched to tenant requirements.
- You have deployment models documented with clear performance characteristics and SLA commitments for each tier.
- You have a deployment model that supports efficient tenant onboarding through configuration rather than infrastructure provisioning.

Common anti-patterns:

- Deploying fully siloed infrastructure for every tenant regardless of their performance requirements, creating resource waste for tenants that would be well-served by pooled resources.
- Using a single pooled deployment without isolation mechanisms, allowing high-volume tenants to consume disproportionate resources and degrade performance for others.
- Skipping tenant tiers with different performance SLAs, treating all tenants identically regardless of business value.

Benefits of establishing this best practice:

- Resource investment stays proportional to tenant value and performance requirements.
- Appropriate isolation mechanisms deliver consistent performance for every tenant.
- Configuration-driven provisioning speeds tenant onboarding.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Define tenant tiers based on performance requirements and business value:

- A standard tier using pooled resources with best-effort performance
- A premium tier using dedicated resources with stronger SLA targets
- (Optional) An enterprise tier with fully isolated infrastructure

For pooled deployments, [Amazon Bedrock AgentCore Runtime](#) provides session-isolated execution that naturally helps prevent cross-tenant interference at the agent execution level. For tenants that need custom container environments, deploy on [Amazon EKS](#) or [Amazon ECS](#) with namespace-level or task-level isolation. For simpler tenant needs such as team-specific chat assistants or enterprise Q&A, [Amazon Quick Suite](#) provides a managed no-code option where business users create and deploy agents without custom infrastructure.

Tenant-aware routing at the API layer uses [Amazon API Gateway](#) with usage plans that enforce per-tenant rate limits and throttling. At the inference layer, Amazon Bedrock's provisioned throughput gives premium tenants reserved capacity while standard tenants use on-demand capacity. Tenant context propagates through the agent stack so every component (runtime, memory, tools) applies tenant-specific configurations. For data isolation, use [Amazon DynamoDB](#) with tenant partition keys for pooled access, or separate tables for siloed tenants. Tenant onboarding automated through [AWS CDK](#) or CloudFormation makes new standard tenants a configuration change rather than an infrastructure provisioning project.

Implementation steps

1. **Define tenant tiers with specific performance SLAs, isolation requirements, and pricing:**
Define standard, premium, and optional enterprise tiers with explicit SLAs and isolation expectations.

2. **Design the deployment architecture for each tier:** Architect pooled (shared AgentCore Runtime, on-demand Amazon Bedrock), premium (dedicated resources, provisioned throughput), and enterprise (fully isolated) deployments.
3. **Implement tenant-aware routing using API Gateway with per-tenant usage plans and rate limits:** Use [Amazon API Gateway](#) usage plans with per-tenant rate and burst limits.
4. **Configure data isolation using DynamoDB tenant partition keys (pooled) or separate tables (siloe):** Enforce data isolation at the data layer with partition keys or separate tables as appropriate for the tier.
5. **Automate tenant onboarding for each tier using CDK/CloudFormation templates:** Template onboarding so new tenants are provisioned through configuration rather than manual infrastructure work.
6. **Monitor per-tenant performance metrics to validate SLA compliance:** Publish per-tenant metrics to CloudWatch and alert when observed performance drifts outside SLA.

Resources

Related best practices:

- [AGENTPERF07-BP02 Implement tenant-aware performance isolation and throttling](#)
- [AGENTPERF02-BP01 Design efficient reasoning pipelines](#)

Related documents:

- [Building multi-tenant architectures for agentic AI on AWS](#)
- [Enforcing tenant isolation, Multi-tenant agentic AI](#)
- [Operationalizing agentic AI on AWS](#)

Related videos:

- [Building multi-tenant SaaS agents with AgentCore \(SAS407\)](#)
- [Transforming from SaaS to multi-tenant agentic SaaS \(SAS304\)](#)
- [Deploy Production-Ready Agents in 22 Minutes with AgentCore Runtime](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon API Gateway](#)
- [Amazon Bedrock](#)
- [Amazon DynamoDB](#)
- [Amazon EKS](#)
- [AWS CDK](#)

AGENTPERF07-BP02 Implement tenant-aware performance isolation and throttling

Trust in a shared agent service is built through consistent, predictable performance for every tenant, even during demand spikes. In pooled multitenant deployments, effective isolation requires throttling at multiple layers (API, inference, memory, and tools), monitoring per-tenant resource consumption, and adaptive fairness mechanisms that distribute shared resources equitably based on current load.

Desired outcome:

- You have per-tenant throttling enforced at every shared resource layer.
- You have tenant resource consumption monitored in real time with alerts for tenants approaching their limits.
- You have graceful throttling that provides clear feedback to throttled tenants.
- You have performance isolation validated through regular load testing that simulates noisy neighbor scenarios.

Common anti-patterns:

- Applying throttling only at the API gateway layer without enforcing limits at downstream shared resources, letting tenants bypass API-level limits through long-running operations.
- Using static throttling limits that don't adapt to current system load, wasting available capacity during low-load periods or failing to protect isolation during high-load periods.
- Throttling all tenants equally regardless of their service tier, failing to honor premium SLAs.

Benefits of establishing this best practice:

- Multi-layer throttling distributes shared resources fairly across tenants.
- Real-time per-tenant consumption metrics support proactive management.
- Per-tenant performance monitoring validates SLA compliance.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Multi-layer throttling enforces tenant limits at every shared resource. At the API layer, [Amazon API Gateway](#) usage plans with per-tenant API keys enforce request rate and burst limits. At the inference layer, tenant-aware request queuing caps concurrent [Amazon Bedrock](#) inference calls per tenant. At the memory and tool layers, per-tenant rate limiting applies to shared endpoints. For agents deployed on [Amazon Bedrock AgentCore Runtime](#), the runtime's session isolation provides natural per-session resource boundaries.

Adaptive throttling adjusts limits based on current system load: during low-load periods, tenants can burst above their baseline limits to use available capacity, and during high-load periods, strict limits protect isolation. Per-tenant [Amazon CloudWatch](#) dashboards and metrics track request volume, inference consumption, latency percentiles, throttle rates, and error rates. Alarms fire when a tenant approaches their limits or when per-tenant latency exceeds SLA thresholds. Regular noisy neighbor testing, simulating high-load scenarios for individual tenants, validates that other tenants' performance stays within SLA bounds.

Implementation steps

1. **Define per-tenant throttling limits for each resource layer:** Set per-tenant limits for API requests per second, concurrent inference calls, memory storage quota, and tool invocations per minute.
2. **Implement API Gateway usage plans with per-tenant API keys and rate/burst limits:** Use [Amazon API Gateway](#) usage plans with per-tenant API keys to enforce rate and burst limits at ingress.
3. **Deploy tenant-aware inference queuing with per-tenant concurrency limits:** Queue [Amazon Bedrock](#) inference calls per tenant so no single tenant can consume all inference capacity.
4. **Configure adaptive throttling that adjusts limits based on current system load:** Allow bursts during low-load periods and enforce strict limits during high-load periods to protect isolation.

5. **Create per-tenant CloudWatch dashboards and configure SLA-based alarms:** Publish per-tenant metrics in [Amazon CloudWatch](#) and alarm on consumption approaching limits or latency exceeding SLA thresholds.
6. **Establish regular noisy neighbor load testing to validate isolation effectiveness:** Schedule noisy neighbor load tests that simulate high-load scenarios for individual tenants and verify others stay within SLA.

Resources

Related best practices:

- [AGENTPERF07-BP01 Design efficient multi-tenant agent deployment models](#)
- [AGENTPERF04-BP02 Implement efficient protocol-based agent communications](#)
- [AGENTSUS01-BP04 Scale cognitive processing pathways appropriately](#)

Related documents:

- [Building multi-tenant architectures for agentic AI on AWS](#)
- [Enforcing tenant isolation, Multi-tenant agentic AI](#)

Related videos:

- [Building multi-tenant SaaS agents with AgentCore \(SAS407\)](#)

Related services:

- [Amazon API Gateway](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [Amazon DynamoDB](#)
- [Amazon Bedrock](#)

Cost optimization

Organizations that treat cost as a first-class design constraint from the start build agentic systems that scale sustainably and deliver measurable business value per dollar spent. Agentic AI introduces cost dynamics that differ fundamentally from traditional cloud workloads, reasoning cycles consume tokens through iterative plan-execute-verify-reflect loops, multi-agent coordination adds multiplicative overhead, and autonomous tool invocations generate unpredictable cost spikes. Effective cost optimization requires designing with cost awareness from inception, right-sizing model and memory capabilities to task requirements, and implementing full visibility to enable data-driven decisions.

Capabilities

- [Reasoning and execution cost optimization](#)
- [Model invocation and token cost optimization](#)
- [Agent memory and state cost management](#)
- [Agent tool serving cost optimization](#)
- [Agent cost visibility and attribution](#)
- [Agent discovery and deployment cost optimization](#)
- [Agent cost governance and continuous optimization](#)

Design principles

In addition to the lens-level design principles, the cost optimization best practices in this lens are represented by at least one of the following principles:

- **Pay only for the reasoning the task requires:** Match model class, context length, and reasoning depth to task complexity instead of provisioning the largest model and longest context for the worst case.
- **Enforce consumption ceilings at every layer:** Token budgets, iteration limits, time bounds, and concurrency caps live as policy at the gateway, runtime, and orchestration tiers so unbounded execution is rejected at the boundary rather than paid for and discovered later.
- **Reuse before you recompute:** Caching of prompts, tool results, retrievals, and intermediate state turns repeat work into lookups. Inference is the most expensive operation in the system; build the architecture around that fact.

- **Attribute spend to the unit that drives it:** Tag every invocation with agent, tenant, session, and workflow so reporting moves from infrastructure-level to per-decision accounting and over-provisioned components become visible.
- **Close the loop between cost data and architecture:** Cost telemetry feeds back into model selection, prompt design, and orchestration choices on a regular cadence. Optimization is continuous, not a one-time review.

Reasoning and execution cost optimization

Teams that initially design cost-aware reasoning patterns can achieve predictable token budgets and avoid the cost growth that can emerge in agentic projects after launch. Agent reasoning cycles consume tokens through iterative plan-execute-verify-reflect loops, and multi-agent coordination adds multiplicative overhead. Unlike traditional applications where compute costs are predictable, agentic systems can accumulate cost in extended reasoning loops or inefficient agent-to-agent communication patterns.

AGENTCOST01: How do you optimize agent reasoning and execution costs?

Capability intent

- Agent reasoning cycles are bounded by explicit termination conditions and confidence-based exits, so token consumption is predictable and proportional to decision complexity.
- Multi-agent coordination scales with task complexity rather than conversation length, because only the minimum context required for each handoff is transmitted between agents.
- Orchestration mechanisms are matched to the determinism of each routing decision, so expensive model invocations are used only where natural language understanding is genuinely required.
- Agent hierarchies are as shallow as the workflow allows, with autonomous workers that complete multi-step sub-tasks without per-step supervisor check-ins.
- Reasoning and coordination costs are instrumented as distinct, observable metrics, and cost-quality baselines feed continuous refinement of thresholds, manifests, and delegation patterns.

Maturity levels

These levels summarize what each stage of maturity looks like for reasoning and execution cost optimization as a whole.

Level	Name	What it looks like
1	Initial	Agents run without explicit termination contracts. Reasoning loops continue until they happen to exit or time out. Multi-agent workflows pass full conversation history at each handoff, and orchestration cost isn't separated from worker cost. Token usage is reviewed only after an unexpected bill or a production incident.
2	Emerging	Teams have adopted basic termination contracts, including iteration caps and session-level token budgets, and tag invocations so orchestration and worker costs can be reported separately. Shared context for collaborating agents is starting to displace per-invocation context relay, but isn't yet the default. Amazon Bedrock AgentCore Observability is enabled for most production agents, and manual reviews of reasoning cost occur at regular intervals.

Level	Name	What it looks like
3	Defined	Cost-quality baselines exist per reasoning phase. Selective reflection is used so full self-correction runs only when initial output quality falls below a threshold. Handoffs follow structured payload schemas, and shared memory through Amazon Bedrock AgentCore Memory is the default for collaborating agents. Orchestration-to-execution token ratios are tracked per workflow, and teams choose AI supervision over rule-based routing only after a determinism analysis.

Level	Name	What it looks like
4	Proactive	Termination conditions, iteration limits, and routing policies are enforced at the control-plane boundary through Amazon Bedrock AgentCore Policy and Amazon Bedrock AgentCore Gateway rather than relying on agent self-restraint. Hybrid supervisor patterns run in production, and plan-then-execute is the default for repeatable workflows . Per-tier cost attribution is automated, with AWS Budgets alerts on orchestration-to-execution ratios and supervisor-to-worker ratios. Tool call efficiency is evaluated in CI/CD through Amazon Bedrock AgentCore Evaluations .

Level	Name	What it looks like
5	Optimized	Termination parameters, manifest compression, and delegation depth are recalibrated continuously from observability data rather than through manual review cycles. Reasoning cost models and supervisor-to-worker ratio targets drive design review for every new workflow. Agent architectures evolve primarily in response to cost-quality telemetry, and the organization contributes reasoning-cost patterns and measurements back to its communities of practice.

Common issues to watch for

- Teams run agents without explicit iteration caps, confidence thresholds, or token budgets, which can leave unbounded reasoning loops undetected until reviewing cost metrics or performance data.
- Multi-agent workflows pass full conversation history between agents at every handoff, so coordination costs scale with conversation length rather than task complexity.
- Routing decisions default to AI supervision even where a simple rule or lightweight classifier would suffice, inflating orchestration cost at every decision point in the workflow.
- Agent hierarchies are deeper than the workflow needs, multiplying model invocations at each delegation and synthesis layer without adding decision quality.
- Aggregate workflow cost is the only metric in use, so orchestration overhead and per-tier cost ratios stay invisible until they are already disproportionate to execution value.

Best practices

- [AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops](#)
- [AGENTCOST01-BP02 Optimize multi-agent collaboration cost through efficient handoff patterns](#)
- [AGENTCOST01-BP03 Implement cost-effective patterns like hybrid supervisor for multi-agent coordination](#)
- [AGENTCOST01-BP04 Design agent hierarchies and delegation patterns that reduce coordination overhead](#)

AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops

Unbounded reasoning loops consume tokens unpredictably and can result in higher than expected token consumption for routine tasks. A bounded reflection pattern gives you predictable token budgets and preserves decision quality.

Desired outcome:

- You have explicit termination conditions for every agent: a maximum iteration count, a confidence threshold, and a per-session token budget.
- You apply reflection selectively, triggering full self-correction only when initial output quality falls below a threshold.
- You track per-cycle token consumption and decision quality so termination parameters can be tuned from data rather than guesswork.

Common anti-patterns:

- Running agents without iteration limits or cost caps, allowing indefinite token consumption without progress toward the task.
- Applying expensive reflection and self-correction to every output, regardless of whether the initial answer was already good.
- Operating without per-cycle token instrumentation, so no one can tell which reasoning phase drives cost.
- Using fixed iteration counts instead of confidence thresholds, which either wastes tokens on unnecessary iterations or cuts off complex reasoning prematurely.
- Building reflection patterns without budget guardrails, so unbounded loops consume tokens before alerts fire.

Benefits of establishing this best practice:

- Predictable token consumption through bounded reasoning cycles with explicit termination conditions.
- Selective reflection preserves decision quality for ambiguous cases while reducing token waste on straightforward tasks.
- Cost-quality baselines reveal which reasoning patterns deliver the best trade-offs, enabling data-driven tuning of thresholds.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Every reflection loop assumes that another iteration will improve the answer more than it costs, which works with ambiguous tasks but often loses value on straightforward ones. Without that contract, agents reflect on every output regardless of whether reflection improves quality. The discipline is to emit a structured confidence signal alongside each action, inspect it in the orchestration layer, and short-circuit the loop when confidence clears a threshold. Otherwise the loop runs until it hits a hard iteration ceiling, which is both the slowest and most expensive outcome for the common case.

Enforcement matters as much as the contract. Iteration caps expressed only in the system prompt can drift past under adversarial inputs or prompt injections. [Amazon Bedrock AgentCore Policy](#) applies Cedar policies at the [Amazon Bedrock AgentCore Gateway](#) boundary, so iteration and token limits are rejected at the traffic layer rather than noticed after they're exceeded. [Amazon Bedrock AgentCore Runtime](#) provides session-isolated execution and consumption-based pricing, so each session carries its own budget and one runaway session doesn't corrupt accounting for others.

Selective reflection separates ambiguity handling from cheaper routine work. Score the initial output against a lightweight rubric, a small model or heuristic, and gate full reflection on that score. Tag reflection outcomes with the task category so you can see where reflection consistently improves quality and where it adds cost with no benefit. Categories that never benefit from reflection should have the trigger disabled entirely. [Amazon Bedrock AgentCore Evaluations](#) supports LLM-as-a-Judge assessment of decision quality, which gives you an objective confidence signal rather than a self-reported one from the agent being evaluated.

The plan, execute, verify, and reflect phases within a reflection cycle have different reasoning intensities. Routing planning and verification to smaller, faster models while reserving the largest

model for execution captures cumulative savings on the frequent low-cost phases, offsetting the higher per-token cost of the infrequent high-intensity phase.

Implementation steps

1. **Define explicit termination conditions per agent:** Set a maximum iteration count, a confidence threshold, and a per-session token budget, and enforce them through [Amazon Bedrock AgentCore Policy](#) Cedar policies at the AgentCore Gateway boundary so enforcement happens at the traffic layer rather than in application code.
2. **Instrument per-cycle token consumption:** Enable [Amazon Bedrock AgentCore Observability](#) to capture per-session token counts through OpenTelemetry, and configure Amazon CloudWatch alarms on anomalous per-cycle patterns.
3. **Establish objective confidence thresholds:** Configure [Amazon Bedrock AgentCore Evaluations](#) to score decision quality through LLM-as-a-Judge, and anchor early-termination thresholds to measured quality rather than self-reported confidence.
4. **Gate reflection on initial output quality:** Score each initial output with a lightweight rubric and trigger the full reflection pass only when the score falls below a configurable threshold, keeping reflection overhead off the straightforward cases.
5. **Recalibrate thresholds on a cadence:** Review cost-quality baselines monthly (or quarterly for stable workloads) and adjust confidence thresholds, iteration limits, and reflection triggers based on the distribution of observed outcomes.

Resources

Related best practices:

- [AGENTCOST01-BP02 Optimize multi-agent collaboration cost through efficient handoff patterns](#)
- [AGENTCOST01-BP03 Implement cost-effective patterns like hybrid supervisor for multi-agent coordination](#)
- [AGENTCOST02-BP01 Architect tiered model selection for cost-performance optimization](#)
- [AGENTCOST07-BP01 Implement automated cost controls with intelligent cutoffs](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Observability](#)

- [Amazon Bedrock AgentCore Policy](#)
- [Amazon Bedrock AgentCore Evaluations](#)
- [Evaluate models in Amazon Bedrock](#)
- [Agentic AI patterns and workflows on AWS](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)

Related videos:

- [AWS 2025 - AgentCore Observability: Monitor and Debug with OpenTelemetry](#)
- [AWS re:Invent 2024 - Balance cost, performance & reliability for AI at enterprise scale \(AIM3304\)](#)
- [AWS re:Invent 2024 - Sustainable and cost-efficient generative AI with agentic workflows \(AIM333\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Runtime tutorials](#)

Related workshops:

- [Diving Deep into Bedrock AgentCore - Evaluations](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTCOST01-BP02 Optimize multi-agent collaboration cost through efficient handoff patterns

In multi-agent systems, the largest hidden cost is redundant context that travels with every handoff. Structured handoff messages and shared memory keep coordination cost proportional to task complexity rather than conversation length.

Desired outcome:

- You have handoff messages carrying only the task specification, relevant facts, and constraints, not full conversation transcripts.
- You have collaborating agents sharing common context through a managed memory layer instead of re-transmitting it on every handoff.
- You track per-handoff and per-workflow coordination costs as distinct metrics.

Common anti-patterns:

- Passing full conversation history in every handoff, causing input token cost to scale with conversation length regardless of relevance to the receiving agent.
- Building deep supervisor hierarchies where multi-level nesting adds orchestration model invocations at each layer, so coordination cost exceeds execution value.
- Skipping shared memory for collaborating agents, re-transmitting common facts in every agent's context window and causing linear cost growth with agent count.
- Running multi-agent workflows without handoff cost tracking, reducing the risk of identification of workflows where coordination overhead has grown disproportionate to the execution work.

Benefits of establishing this best practice:

- Coordination overhead stays proportional to task complexity rather than conversation length or agent count.
- Shared memory removes redundant context transmission, reducing per-handoff token cost.
- Per-handoff cost visibility enables data-driven tuning of multi-agent interaction patterns.

Level of risk exposed if this best practice is not established: High

Implementation guidance

The most expensive thing an agent can send is context the receiver already has (or doesn't need). Every handoff that copies the full conversation across the boundary pays again for information that never changed. Treat handoff messages as structured summaries containing the task specification, relevant facts, and constraints the worker must respect. [Amazon Bedrock AgentCore Memory](#) enables write-once, read-many patterns where one agent stores a fact under a session ID and actor ID, and every collaborator reads it without re-embedding it in their own prompt. [Amazon Bedrock AgentCore Gateway](#) adds the corresponding discipline on the tool side, using

MCP-compatible Semantic Tool Selection to present only tools relevant to the current intent rather than the full catalog.

Context distillation means that a small model call or a Lambda function can compress incoming context into the minimum sufficient information for the next agent's task before the handoff crosses the boundary. The cost of the distillation call is typically less than the cost of repeatedly transmitting untrimmed context through deeper workflows.

Every supervisor-worker layer adds at least one inference for delegation and one for synthesis. Hierarchies deeper than three levels compound that overhead quickly, and most deep hierarchies can be flattened by replacing intermediate supervisors with direct worker-to-worker communication through the AgentCore Runtime Agent-to-Agent protocol. The diagnostic metric is the orchestration-to-execution token ratio. Supervisors should consume no more than 20% of total workflow tokens, leaving 80% for workers doing execution. A ratio that drifts above 20% means coordination has grown disproportionate to work.

Visibility is a prerequisite for these patterns. [Amazon Bedrock AgentCore Observability](#) provides distributed tracing so agent-to-agent communication costs appear as their own category rather than hidden inside aggregate workflow cost. [Amazon Bedrock AgentCore Evaluations](#) runs in real time against live tool-call traces and as offline test suites in CI/CD pipelines, so redundant or unnecessary invocations are caught early.

Implementation steps

1. **Design structured handoff messages:** Replace full conversation history with a summary object containing the task specification, relevant facts, and the constraints the receiving agent must respect. Version the message schema so receivers can reject malformed handoffs.
2. **Insert context distillation at boundaries:** Add a small-model call or Lambda function that extracts minimum sufficient context before each handoff, so input tokens at transitions reflect current task needs rather than accumulated history.
3. **Configure shared memory with ownership rules:** Provision [Amazon Bedrock AgentCore Memory](#) accessible to all collaborating agents, and document which agent owns writes to each namespace so shared state has a clear provenance.
4. **Flatten deep hierarchies:** Audit multi-agent workflows for supervisor-worker depth greater than three levels, and replace intermediate supervisors with direct worker-to-worker communication through the AgentCore Runtime Agent-to-Agent protocol where the routing can be made explicit.

5. **Expose specialized agents through Gateway:** Publish agents as tools through [Amazon Bedrock AgentCore Gateway](#) MCP server capabilities, and turn on Semantic Tool Selection so collaborating agents see only tools relevant to the current request.
6. **Evaluate tool-call efficiency in CI and in production:** Run [Amazon Bedrock AgentCore Evaluations](#) against live tool-call traces to flag inefficient usage at runtime, and against offline test suites in the CI/CD pipeline to catch regressions before deployment.
7. **Track the orchestration-to-execution ratio:** Tag every invocation with workflow-id and agent-role, build CloudWatch dashboards that display the supervisor-to-worker token ratio per workflow, and configure AWS Budgets alerts when orchestration overhead exceeds 20% of total workflow token cost.

Resources

Related best practices:

- [AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops](#)
- [AGENTCOST01-BP03 Implement cost-effective patterns like hybrid supervisor for multi-agent coordination](#)
- [AGENTCOST01-BP04 Design agent hierarchies and delegation patterns that reduce coordination overhead](#)
- [AGENTCOST05-BP02 Implement distributed cost tracing for multi-agent workflows](#)

Related documents:

- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock AgentCore Evaluations](#)
- [Agentic AI patterns and workflows on AWS](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Memory](#)

- [AWS re:Invent 2024 - Balance cost, performance & reliability for AI at enterprise scale \(AIM3304\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Multi-agent tutorials](#)
- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Evaluations tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTCOST01-BP03 Implement cost-effective patterns like hybrid supervisor for multi-agent coordination

Many multi-agent workflows pay for AI reasoning at the coordination layer for decisions that rules could make at no additional charge. Matching each orchestration decision to the cheapest mechanism capable of handling it removes that hidden cost.

Desired outcome:

- You select orchestration patterns from workflow determinism analysis rather than defaulting to AI supervision.
- You have deterministic routing running without model invocations, and AI supervisors reserved for genuinely ambiguous cases that need natural language understanding.
- You track orchestration cost separately from worker cost and maintain documented pattern-selection criteria.

Common anti-patterns:

- Using AI supervisors for deterministic workflows, invoking expensive foundation models for routing decisions that straightforward rules handle.
- Defaulting to AI supervision without evaluating whether the routing logic follows explicit rules.
- Tracking only aggregate workflow cost without decomposing orchestrator compared to worker spend, which hides disproportionate coordination overhead.

Benefits of establishing this best practice:

- Rule-based routing handles deterministic branches without model invocations, reducing per-routing-decision cost to near zero.
- Hybrid patterns match each routing decision to the cheapest capable mechanism.
- Documented pattern-selection criteria help prevent over-provisioning AI supervision for new workflows.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Workflow determinism is a property, not an assumption. At every orchestration point, the routing decision is either a selection across a finite, enumerable set of conditions (task type, output classification, error code) or a judgment call across an open-ended input space. The first class costs nothing to route with rules, and the second requires model reasoning. Most multi-agent workflows contain both, but teams often pay for AI supervision across the whole workflow because the pattern defaults that way. Conducting the determinism analysis up front is the difference between spending model tokens on routing that a conditional could handle and spending them only where the input genuinely demands natural-language interpretation.

Enforcement happens outside the agent code. [Amazon Bedrock AgentCore Policy](#) runs Cedar policies at the [Amazon Bedrock AgentCore Gateway](#) boundary, applying deterministic routing rules based on task attributes, user identity, or tool requirements without invoking an inference. Worker agents deploy on [Amazon Bedrock AgentCore Runtime](#) at the leaf nodes where reasoning actually happens. Keeping routing and reasoning on separate rails lets each one evolve independently and be monitored on its own metrics.

For partially deterministic routing, a tiered hybrid pattern helps you align costs. A lightweight classifier (a small Amazon Bedrock model or a rule-based heuristic) attempts rule-based routing first and escalates to the full AI supervisor only when its confidence falls below a configured threshold. The escalation rate is the signal for whether the tier is tuned correctly. If the rate is too high, the classifier needs refinement. If it is too low, the supervisor is over-provisioned and the classifier can absorb more cases.

Quantitative thresholds provide rule-based routing for workflows with fewer than ten deterministic branches, a lightweight classifier for routing across ten to fifty categories, and AI supervisors only for unbounded category spaces that require natural-language understanding. The orchestration

overhead ratio (supervisor tokens divided by total workflow tokens) is the ongoing diagnostic. When it drifts above the baseline, the pattern needs reassessment, not a larger budget.

Implementation steps

1. **Conduct workflow determinism analysis:** At each orchestration point in the workflow, classify the routing decision as fully deterministic, partially deterministic, or open-ended, and record the rationale as an architectural decision record so downstream reviewers can audit why each pattern was chosen.
2. **Apply Cedar policies for deterministic routing:** Configure [Amazon Bedrock AgentCore Policy](#) with Cedar policies at [Amazon Bedrock AgentCore Gateway](#) for every fully deterministic branch, so these routing decisions run without model invocations.
3. **Insert a lightweight classifier for partially deterministic routing:** Deploy a small model or rule-based heuristic that attempts rule-based routing first and escalates to a full AI supervisor only when its confidence falls below a configurable threshold, and log the escalation rate as a tuning signal.
4. **Separate orchestration cost from worker cost:** Configure [Amazon Bedrock AgentCore Observability](#) to attribute tokens to orchestrator and worker tiers separately, calculate the orchestration overhead ratio per workflow, and alert when the ratio drifts above the baseline recorded for that pattern.

Resources

Related best practices:

- [AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops](#)
- [AGENTCOST01-BP02 Optimize multi-agent collaboration cost through efficient handoff patterns](#)
- [AGENTCOST01-BP04 Design agent hierarchies and delegation patterns that reduce coordination overhead](#)
- [AGENTCOST05-BP02 Implement distributed cost tracing for multi-agent workflows](#)

Related documents:

- [Amazon Bedrock AgentCore Policy](#)
- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Observability](#)

- [Agentic AI patterns and workflows on AWS](#)

Related videos:

- [AWS re:Invent 2024 - Balance cost, performance & reliability for AI at enterprise scale \(AIM3304\)](#)
- [AWS re:Invent 2024 - Sustainable and cost-efficient generative AI with agentic workflows \(AIM333\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Policy tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTCOST01-BP04 Design agent hierarchies and delegation patterns that reduce coordination overhead

Supervisor cost in agent hierarchies grows with the verbosity of capability descriptions and the frequency of check-ins. Compact manifests and autonomous workers keep coordination cost proportional to workflow complexity rather than step count.

Desired outcome:

- Your agent hierarchies use the shallowest orchestration structure capable of managing the workflow.
- You have supervisor agents operating on compressed capability manifests that minimize input tokens per routing decision.
- Your worker agents complete multi-step sub-tasks autonomously, escalating to supervisors only for task assignment and result validation.
- You track orchestrator cost as a distinct category with a target supervisor-to-worker cost ratio.

Common anti-patterns:

- Including verbose natural-language descriptions of every worker's capabilities in routing prompts, which inflates token cost that then scales linearly with worker count.
- Requiring supervisor check-ins after each sub-step, which multiplies coordination overhead when workers could complete multi-step work autonomously.
- Tracking only aggregate workflow cost without decomposing orchestrator compared to worker expense, so disproportionate coordination overhead hides in the total.

Benefits of establishing this best practice:

- Compressed capability manifests reduce supervisor input-token cost per routing decision.
- Autonomous workers remove supervisor round-trips for intermediate decisions.
- Per-tier cost attribution surfaces optimization opportunities where coordination overhead exceeds execution value.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Supervisor cost has two main drivers: how expensive each routing decision is and how many times routing happens.

The first is controlled by manifest size. Supervisors that describe workers in paragraphs of natural language pay for those paragraphs on every routing call, and that cost scales linearly with worker count. Short, structured capability manifests (description, input schema, output schema, under 200 tokens each) cut this cost without sacrificing routing quality, because the supervisor doesn't need prose to choose between workers that have distinct schemas.

The second is controlled by context relay. When context flows from parent to worker through the supervisor, every byte of that context is transmitted twice: once into the supervisor, and once into the worker as part of the routing response. [Amazon Bedrock AgentCore Memory](#) removes that doubling by letting workers read shared context directly from memory using the session's actor ID and session ID, so the supervisor only routes rather than relays. [Amazon Bedrock AgentCore Gateway](#) reduces it further by supporting runtime tool discovery through Model Context Protocol, so the supervisor prompt doesn't need to enumerate every tool the workers can call. [Amazon Bedrock AgentCore Policy](#) controls which tools each worker is allowed to invoke autonomously, making it safe to shift decisions downward without losing governance.

Workers designed with sufficient tool autonomy and clear success criteria can complete multi-step sub-tasks, returning a single structured result with a confidence score. The supervisor then makes an efficient accept-or-reject decision rather than re-reasoning from scratch at each intermediate step. For workflows with repeatable decomposition patterns, a plan-then-execute approach compresses this further, where one supervisor invocation generates the full task plan, then workers execute the plan without further supervision.

Track the supervisor-to-worker cost ratio. Set a target (for example, supervisor tokens no more than 15% of worker tokens) and alert when it is exceeded. A breach typically signals that manifest compression, worker autonomy, or plan-then-execute adoption is needed.

Implementation steps

1. **Compress worker capability descriptions:** Replace natural-language capability descriptions with structured manifests (description, input schema, output schema) under 200 tokens each, and use [Amazon Bedrock AgentCore Gateway](#) runtime tool discovery to avoid listing tools in the supervisor prompt.
2. **Redesign workers for autonomous multi-step completion:** Give each worker sufficient tool autonomy and clear success criteria to complete its sub-task end-to-end, and require the worker to emit a confidence score in every response so the supervisor can make accept-or-reject decisions without re-reasoning.
3. **Apply policy and shared memory for direct context access:** Configure [Amazon Bedrock AgentCore Policy](#) through Gateway to enforce worker tool-access boundaries, and provision [Amazon Bedrock AgentCore Memory](#) so workers read shared context directly instead of receiving it relayed through the supervisor.
4. **Track supervisor-to-worker cost ratio:** Configure [Amazon Bedrock AgentCore Observability](#) to attribute tokens per tier, build Amazon CloudWatch dashboards showing the supervisor-to-worker ratio per workflow, and alert when the ratio exceeds a 15% target.

Resources

Related best practices:

- [AGENTCOST01-BP02 Optimize multi-agent collaboration cost through efficient handoff patterns](#)
- [AGENTCOST01-BP03 Implement cost-effective patterns like hybrid supervisor for multi-agent coordination](#)
- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)

- [AGENTCOST05-BP02 Implement distributed cost tracing for multi-agent workflows](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Policy](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Agentic AI patterns and workflows on AWS](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Gateway](#)
- [AWS 2025 - AgentCore Observability: Monitor and Debug with OpenTelemetry](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Multi-agent tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Model invocation and token cost optimization

Teams that right-size model selection to task complexity and optimize token consumption per invocation can improve their cost reductions without sacrificing agent decision quality. Foundation model costs can dominate agentic AI budgets, with agents potentially invoking expensive models for simple tasks or consuming excessive tokens through verbose reasoning.

AGENTCOST02: How do you optimize agent model invocation and token consumption costs?

Capability intent

- Model selection matches task complexity, with routine classification and formatting handled by cost-efficient models and premium models reserved for reasoning that genuinely needs them.
- Prompts, tool descriptions, and output constraints are kept at the minimum size required to maintain decision quality across planning, execution, and reflection phases.
- Repeated reasoning and stable context are served from caches rather than regenerated, so equivalent requests don't pay the full inference cost twice.
- High-volume recurring tasks run on specialized or customized models whose one-time training cost amortizes against sustained per-invocation savings.
- Cost-per-correct-response, token consumption, cache hit rate, and cascade escalation rate are tracked per model tier and fed back into routing, caching, and customization decisions.

Maturity levels

These levels summarize what each stage of maturity looks like for model invocation and token cost optimization as a whole.

Level	Name	What it looks like
1	Initial	Teams invoke a single default foundation model for every agent task regardless of complexity, and prompts, tool lists, and outputs grow without review. No prompt caching, semantic caching, or model customization is in place. Cost surprises are attributed to traffic growth rather than investigated.
2	Emerging	A documented model routing policy maps task types to specific model tiers, and environments are aligned to the appropriate Amazon

Level	Name	What it looks like
		Bedrock on-demand pricing tier (Flex, Standard, Priority). Prompt compression, explicit output length constraints, and Amazon Bedrock prompt caching for stable system prompts are applied to the highest-volume agents. Token consumption is reviewed periodically against aggregate cost dashboards.
3	Defined	A pre-classifier routes every invocation to a model tier based on task complexity, and model cascading with confidence-based escalation is the default pattern for non-trivial reasoning. Semantic caching and plan template reuse run alongside prompt caching, with TTLs differentiated by task freshness. Token consumption, cache hit rates, and cascade escalation rates are tracked per reasoning phase using Amazon Bedrock AgentCore Observability , and Bedrock batch inference is used for non-time-sensitive work.

Level	Name	What it looks like
4	Proactive	Routing, caching, and guardrails are automated. Amazon Bedrock AgentCore policies enforce model tier limits, cache policies are event-driven, and fallback chains handle timeouts without same-tier retries. High-volume task categories run on distilled or parameter-efficient fine-tuned models in production, validated through A/B testing with Amazon Bedrock AgentCore Evaluations and served through Custom Model Import on AgentCore Runtime. Cost-per-correct-response is a tracked service-level metric and drives quarterly refresh decisions.

Level	Name	What it looks like
5	Optimized	Model portfolios are continuously optimized . Complexity classifiers, cascade thresholds, cache similarity thresholds, and customization refresh cadences are all tuned from production telemetry . Customized models are refreshed on a continuous pipeline with automated training data extraction, evaluation, and promotion gates. The organization drives cost-per-outcome improvements quarter over quarter while maintaining or raising decision quality baselines, and contributes patterns and benchmarks back to the broader practitioner community.

Common issues to watch for

- Every agent invocation defaults to the most capable general-purpose model, resulting in higher inference costs for routine classification and formatting that could use cost-effective alternatives.
- System prompts and tool catalogs accumulate without review, so the fixed input-token tax paid on every invocation rises silently across months even when traffic is flat.
- Caching is treated as a one-time enablement rather than an ongoing discipline, so stale invalidation policies, weak similarity thresholds, or drifting cacheable prefixes erode hit rates and savings without anyone noticing.

- Model customization is pursued on low-volume or vanity workloads rather than on high-volume recurring tasks where training cost can amortize, leaving the investment permanently underwater.
- Cost signals are tracked only at aggregate level, so teams can't see which tier, reasoning phase, or cache layer is driving spend and can't prioritize optimization work against evidence.

Best practices

- [AGENTCOST02-BP01 Architect tiered model selection for cost-performance optimization](#)
- [AGENTCOST02-BP02 Cost optimize token consumption through efficient prompt engineering](#)
- [AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations](#)
- [AGENTCOST02-BP04 Implement model customization for long-term cost reduction](#)

AGENTCOST02-BP01 Architect tiered model selection for cost-performance optimization

Running every agent task on the largest available model inflates inference cost by an order of magnitude for work that smaller models handle correctly. Match each task to the cheapest model capable of acceptable quality, and escalate only when confidence drops.

Desired outcome:

- You have agent tasks classified into complexity tiers, with a documented routing policy mapping each tier to a specific foundation model.
- You have cascading patterns that escalate to higher-cost models only when a lower tier's confidence falls below threshold.
- You track cost-per-correct-response across tiers and refresh routing decisions with the data rather than with intuition.

Common anti-patterns:

- Using the largest available model for all agent tasks without assessing task complexity, inflating inference costs for routine operations.
- Hard-coding static model assignments without confidence-based escalation, which either over-provisions routine tasks or under-provisions complex edge cases.

- Tracking aggregate costs without decomposing agent performance by model tier, hiding opportunities to shift workloads to cheaper models.
- Failing to monitor customized model performance after switching to a smaller tier, allowing cost savings to mask hidden quality degradation.

Benefits of establishing this best practice:

- Tiered selection reserves expensive models for genuinely complex reasoning and routes routine tasks to cost-effective alternatives.
- Model cascading minimizes premium model invocations through confidence-based escalation.
- Specialized models for domain-specific tasks deliver higher accuracy at lower cost than general-purpose alternatives.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Task complexity is an important property to measure. At every agent invocation, the reasoning you need is either lightweight (classification, format conversion, intent extraction), moderate (multi-step reasoning, summarization), or genuinely complex (open-ended analysis, multi-constraint optimization). These three classes map to different price points across the [Amazon Bedrock](#) model catalog, and treating them identically means you pay the complex-class price for every low-complexity task. Classifying upfront and routing accordingly is where most of the cost headroom sits.

A lightweight pre-classifier gives you that routing decision without invoking the main model first. Rule-based heuristics or a small model can analyze request characteristics like input length, structured or unstructured format, constraint count, and reasoning depth, assigning scores that map to tier thresholds (for example, below 0.3 for simple, 0.3 to 0.7 for moderate, above 0.7 for complex). The pre-classifier must cost less than the tier price differential to produce net savings on first-attempt routing. For multimodal tasks the principle extends further. Route document extraction to Amazon Bedrock Data Automation and audio interactions to Amazon Nova Sonic rather than sending raw images or audio through expensive general-purpose vision models.

Model cascading is a fallback mechanism when the classifier is uncertain. Have the lower-tier model return a structured response with a self-assessed confidence score and escalate to the next tier only when confidence falls below a threshold. Primary, secondary, and tertiary fallback chains

catch timeouts and failures by moving up a tier rather than retrying the same one, improving completion rates without retry waste. [Amazon Bedrock AgentCore Runtime](#) is designed to support multiple frameworks and LLM providers, and [Amazon Bedrock AgentCore Policy](#) enforces guardrails that help prevent expensive model calls when task complexity doesn't justify the cost.

Pricing tier is independent of model size. [Amazon Bedrock capacity, limits, and cost optimization](#) documents Flex for development and testing at the lowest per-token cost, Standard for production, and Priority only for latency-sensitive user-facing interactions where throttling risk must be minimized. Batch inference offers up to 50% savings for non-time-sensitive workloads like report generation, training data preparation, or offline evaluation. For consistent high-volume traffic, Reserved Tier commitments provide 30 to 50% savings against on-demand pricing. With [Amazon Bedrock AgentCore Evaluations](#), you can benchmark multiple model options against your actual task distribution, measuring cost-per-correct-response and refreshing the routing policy quarterly as new models become available.

Implementation steps

1. **Classify agent tasks into complexity tiers:** Document a model routing policy mapping each tier (simple, moderate, and complex) to a specific [Amazon Bedrock](#) model, and commit the policy as an architectural decision record so downstream reviewers can audit the rationale.
2. **Select pricing tier per environment:** Use Flex for development and testing, Standard for production, and Priority only for latency-sensitive user-facing agents, and evaluate [Amazon Bedrock Reserved Tier](#) commitments for consistent high-volume workloads.
3. **Insert a task complexity pre-classifier:** Deploy rule-based heuristics or a small-model call that scores each request on input length, structure, constraint count, and reasoning depth before the main invocation, and make sure the classifier costs less than the tier price differential.
4. **Implement model cascading on confidence:** Have each lower-tier response include a self-assessed confidence score, and escalate to the next tier when confidence falls below the configured threshold rather than retrying at the same tier.
5. **Configure fallback chains per task category:** Define primary, secondary, and tertiary model options, with automatic escalation on timeout or failure instead of retry, so transient failures move up a tier rather than repeating the same cost.
6. **Route non-time-sensitive tasks to batch inference:** Use [Amazon Bedrock batch inference](#) for report generation, data enrichment, and offline evaluation to capture up to 50% savings over on-demand pricing.

7. **Benchmark specialized compared to general-purpose models:** Run [Amazon Bedrock AgentCore Evaluations](#) on your actual task distribution, measuring cost-per-correct-response so routing choices are grounded in outcome data.
8. **Review routing policies quarterly:** Use AWS Cost Explorer and Amazon CloudWatch dashboards to inspect observed escalation rates, and adjust tier assignments when cascade escalation patterns indicate mis-tuned thresholds.

Resources

Related best practices:

- [AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops](#)
- [AGENTCOST01-BP04 Design agent hierarchies and delegation patterns that reduce coordination overhead](#)
- [AGENTCOST02-BP02 Cost optimize token consumption through efficient prompt engineering](#)
- [AGENTCOST02-BP04 Implement model customization for long-term cost reduction](#)
- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)

Related documents:

- [Amazon Bedrock AgentCore Evaluations](#)
- [Effective cost optimization strategies for Amazon Bedrock](#)
- [Use Amazon Bedrock Intelligent Prompt Routing for cost and latency benefits](#)
- [Optimizing cost for using foundational models with Amazon Bedrock](#)
- [Economics for agentic AI on AWS](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)
- [Agentic AI patterns and workflows on AWS](#)
- [Amazon Bedrock model pricing](#)
- [Amazon Bedrock capacity, limits, and cost optimization](#)
- [Amazon Bedrock batch inference](#)

Related videos:

- [AWS re:Invent 2024 - Balance cost, performance & reliability for AI at enterprise scale \(AIM3304\)](#)

- [AWS re:Invent 2024 - Mastering model choice: The 3-step Amazon Bedrock advantage \(AIM391\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Evaluations tutorials](#)

Related tools:

- [Strands Agents Model Providers](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [AWS Cost Explorer](#)

AGENTCOST02-BP02 Cost optimize token consumption through efficient prompt engineering

Every token in a system prompt is paid for on every invocation, so prompt bloat compounds linearly with traffic. Compressing system prompts, tool descriptions, and output formats makes the fixed cost of each agent call proportional to the decision it has to make, not the verbosity of its instructions.

Desired outcome:

- You have agent system prompts compressed to the minimum tokens needed for accurate task completion.
- You have tool descriptions presented dynamically based on the current task rather than transmitted in full on every call.
- You constrain output length explicitly so verbose responses don't compound across multi-turn reasoning.
- You version prompts and track cost-per-task per version so efficiency changes are measurable over time.

Common anti-patterns:

- Writing verbose system prompts with lengthy persona descriptions and redundant explanations, inflating the fixed token cost on every invocation.
- Including every tool description in every invocation regardless of task relevance, which inflates input tokens when only a subset of tools applies.
- Allowing unconstrained output length without formatting directives, enabling verbose responses that compound across multi-turn reasoning cycles.
- Treating prompts as uncontrolled strings rather than versioned artifacts, so regressions in token efficiency go unnoticed until a billing review.

Benefits of establishing this best practice:

- Fixed-cost reductions from prompt compression compound across every agent invocation in high-volume deployments.
- Dynamic tool loading transmits only task-relevant tools, reducing tool description overhead in proportion to catalog size.
- Prompt versioning with token tracking makes compression an ongoing, measurable practice rather than a one-off cleanup.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The system prompt is the largest fixed cost in every model invocation, which means every unnecessary sentence is paid for again each time the agent is called. Start by auditing prompts with [Amazon Bedrock AgentCore Observability](#) to measure token footprints, then compress systematically. Replace verbose instructions with structured directives, tighten role definitions, and remove redundant explanations.

Tool descriptions behave the same way. If the prompt carries the full tool catalog even when only three tools are relevant, you are paying for the rest of the catalog on every call. [Amazon Bedrock AgentCore Gateway](#) uses MCP-based Semantic Tool Selection to present only the tools relevant to the current intent, and [Amazon Bedrock AgentCore Memory](#) manages conversation state across multi-turn sessions so you don't pay for manual history concatenation.

In your context window, allocate tokens across the system prompt (20 to 30%), user context (30 to 40%), few-shot examples (10 to 20%), and agent scratchpad (20 to 30%), and adjust based on your agent's reasoning patterns. For few-shot examples in particular, test whether the model performs well zero-shot before paying for examples on every call. When examples are needed, identify the minimum count that maintains task accuracy against the quality baseline. Two or three examples are typically enough, and dynamic example selection from a semantic index keeps only the relevant ones in context.

Output length is the last thing to adjust. Explicit formatting directives in the system prompt (response structure, maximum length) directly control output token costs. Treat prompts as versioned artifacts. Record token count, task success rate, and cost-per-task for each version using AgentCore Observability telemetry, and establish a monthly review cadence that tracks cumulative savings in AWS Cost Explorer.

Implementation steps

1. **Audit and compress system prompts:** Use [Amazon Bedrock AgentCore Observability](#) to measure token footprints for every prompt, then apply compression to minimize prompt size while maintaining decision accuracy.
2. **Present tools dynamically through Gateway:** Use [Amazon Bedrock AgentCore Gateway](#) Semantic Tool Selection to present only relevant tools per invocation, and compress tool descriptions to tool name, one-sentence description, and concise parameter schema.
3. **Constrain output length:** Add explicit output formatting directives to the system prompt and configure model-specific token limit parameters to enforce hard caps on response size.
4. **Use managed memory for multi-turn context:** Configure [Amazon Bedrock AgentCore Memory](#) so conversation state is maintained automatically instead of re-transmitted as full history.
5. **Reduce few-shot example overhead:** Evaluate whether each agent task needs examples at all, then reduce to the minimum effective count of two to three. Load examples dynamically from an Amazon S3-hosted library using semantic similarity retrieval.
6. **Version prompts and track efficiency:** Record token count and task success rate per prompt version, and establish a monthly optimization review cadence that tracks cumulative savings against cost-per-task targets.

Resources

Related best practices:

- [AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops](#)
- [AGENTCOST02-BP01 Architect tiered model selection for cost-performance optimization](#)
- [AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Memory](#)
- [Economics for agentic AI on AWS](#)

Related videos:

- [AWS re:Invent 2024 - Sustainable and cost-efficient generative AI with agentic workflows \(AIM333\)](#)
- [Strands Tools: Building Custom AI Agents with Python](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Runtime tutorials](#)

Related tools:

- [Strands Agents Custom Tools](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Cost Explorer](#)

AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations

Agents repeat work constantly: identical prompts, semantically equivalent requests, the same planning steps across similar tasks. Caching at the prompt, semantic, and plan-template layers changes repetition from a recurring expense into a one-time cost paid on the first invocation.

Desired outcome:

- You have prompt caching enabled for stable system prompts so the cacheable prefix is reused across invocations at reduced rates.
- You have a semantic cache that serves responses for functionally equivalent requests above a configurable similarity threshold.
- You have plan templates cached and instantiated for recurring task patterns rather than regenerated each time.
- You track cache hit rates and cost savings per caching layer.

Common anti-patterns:

- Transmitting identical system prompts and tool descriptions on every invocation at full input token cost rather than cached prefix rates.
- Using exact-match lookups when functionally equivalent requests use different wording, causing cache misses on semantically identical tasks.
- Applying one cache TTL across all task types without distinguishing static reference data from time-sensitive information, returning stale responses that degrade quality.
- Deploying customized models without monitoring cache-assisted performance, missing opportunities to validate that expected cost reductions actually materialize.

Benefits of establishing this best practice:

- Prompt caching reduces input token costs by reusing cached system instructions across invocations at reduced rates.
- Semantic caching helps prevent redundant reasoning by serving cached responses for functionally equivalent tasks.
- Plan template reuse reduces model invocations for the planning phase of recurring task patterns.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Caching for agents works at three distinct layers, and each layer has a different failure mode.

[Amazon Bedrock prompt caching](#) is the highest-impact layer for agents with large stable system prompts. Amazon Bedrock stores the key-value state of the cached prefix and reuses it at reduced rates. Design so that the cacheable prefix (system prompt, tool descriptions) is stable across invocations, because any dynamic content mixed in invalidates the cache. Refactor to move user-specific or session-specific content out of the cacheable prefix.

Semantic caching addresses the idea that two requests that mean the same thing are rarely identical in wording. Generate an embedding of each incoming request with a lightweight model and query [Amazon OpenSearch Service Serverless](#) for similar prior requests above a configurable threshold such as cosine similarity greater than 0.95. The threshold helps you tune, as higher values reduce false positives but lower hit rates, and the right value depends on how much response variance your agent tolerates. Store cache entries with TTLs calibrated to each task type's freshness requirements, so reference data cached for hours doesn't pollute tasks that need up-to-date market or inventory information.

Don't overlook plan template caching. Agent planning outputs are highly repeatable for recurring task patterns, like an onboarding checklist, a support triage decomposition, or a reporting workflow plan. Store these plans keyed by task type and input parameter signature, and instantiate cached templates with current parameters rather than regenerating new plans each time.

[Amazon Bedrock AgentCore Memory](#) manages conversation state by extracting and persisting key information, reducing input token costs from repeated history transmission.

Cache correctness depends on invalidation. Event-driven invalidation purges stale entries the moment source data changes, which is what makes aggressive caching safe for moderately volatile data. Measure impact with AWS Cost Explorer and Amazon CloudWatch integrated with [Amazon Bedrock AgentCore Observability](#), and alarm when hit rates fall below targets.

Implementation steps

1. **Enable prompt caching for stable prefixes:** Turn on [Amazon Bedrock prompt caching](#) for agents with system prompts larger than 1,000 tokens, and refactor the prompt to move dynamic content out of the cacheable prefix.

2. **Deploy a semantic cache layer:** Stand up an OpenSearch Serverless index with embedding-based similarity, configure similarity thresholds per task type, and set per-task TTLs. Accept quantization only when accuracy loss remains below two percent on task success rate.
3. **Cache plan templates:** Key plan templates by task type and input parameter signature, and perform a pre-invocation lookup before generating a new plan.
4. **Use managed memory for session state:** Configure [Amazon Bedrock AgentCore Memory](#) session identifiers so multi-turn conversation state is maintained without manual history concatenation.
5. **Design event-driven invalidation and monitor hit rates:** Wire event-driven cache invalidation to source data changes, and create CloudWatch dashboards that display hit rates across prompt, semantic, and plan-template caches with alarms when hit rates fall below target.

Resources

Related best practices:

- [AGENCYCOST02-BP01 Architect tiered model selection for cost-performance optimization](#)
- [AGENCYCOST02-BP02 Cost optimize token consumption through efficient prompt engineering](#)
- [AGENCYCOST03-BP01 Design cost-effective retrieval systems with tiered memory](#)

Related documents:

- [Amazon Bedrock Prompt Caching](#)
- [Effectively use prompt caching on Amazon Bedrock](#)
- [Optimize LLM response costs and latency with effective caching](#)
- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Runtime](#)
- [Economics for agentic AI on AWS](#)

Related videos:

- [AWS re:Invent 2024 - Balance cost, performance & reliability for AI at enterprise scale \(AIM3304\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Memory tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon OpenSearch Service](#)
- [Amazon CloudWatch](#)

AGENTCOST02-BP04 Implement model customization for long-term cost reduction

Customizing smaller models for a high-volume recurring task can optimize per-invocation costs into a one-time training expense that amortizes across every future call. The math only works when volume and task stability are high enough to justify the investment, so the decision needs to start with a break-even calculation, not an enthusiasm for fine-tuning.

Desired outcome:

- You have specialized models handling high-volume recurring tasks at materially lower per-invocation cost than general-purpose foundation models.
- You have a customization pipeline that captures decision patterns from production and refreshes models on a scheduled cadence.
- You validate decision quality with A/B testing against foundation models before routing production traffic to a customized model.
- You track inference cost savings and decision quality side by side so positive ROI is provable rather than assumed.

Common anti-patterns:

- Fine-tuning on synthetic data that misrepresents production task distributions, causing underperformance that offsets cost savings through lower task completion rates.
- Applying customization to low-volume task categories where training costs exceed projected inference savings, wasting effort on optimization that doesn't reach positive ROI.
- Treating customization as a one-time project without continuous adaptation, allowing specialized models to drift as workload patterns change.

- Routing production traffic to customized models without A/B testing against foundation models, risking quality degradation that undermines cost savings.
- Deploying customized models without instrumenting inference latency, token costs, and quality metrics, reducing the risk of validation that the expected cost reduction materialized.

Benefits of establishing this best practice:

- Fine-tuned smaller models achieve comparable accuracy at lower per-invocation cost through reduced token consumption and faster inference.
- One-time training costs amortize across thousands of invocations, delivering compounding returns for high-volume tasks.
- Continuous adaptation pipelines keep specialized models aligned with evolving workload patterns rather than decaying silently.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Calculate current monthly inference cost for the target task category using [Amazon Bedrock AgentCore Observability](#), estimate the reduction from a smaller customized model, and compare against one-time customization costs plus ongoing refresh. When monthly inference costs exceed \$500 and task volume exceeds 10,000 invocations per month, customization typically reaches break-even within 6 to 12 months. Make the break-even explicit: (one-time training cost + quarterly refresh cost × planning horizon in quarters) divided by monthly inference savings. For a \$5,000 training run that saves \$400 per month, break-even lands at month 13, which is acceptable for workloads with multi-year lifespans but not for experimental projects.

[Knowledge distillation](#) transfers capability from a large teacher model to a smaller student model at lower per-invocation cost. The training data should come from production invocation logs filtered for high-confidence, successful completions. Parameter-efficient fine-tuning methods like QLoRA quantize base model weights to four-bit precision and train only adapter parameters, making single-GPU fine-tuning viable for smaller teams. [Amazon Bedrock model customization](#) jobs and [Amazon SageMaker AI AI Training Jobs](#) with QLoRA support fine-tuning without managing training infrastructure, and [Amazon Bedrock Custom Model Import](#) brings the results into Amazon Bedrock for serving.

Validation helps prevent quality regressions that occur from these cost optimizations. With [Amazon Bedrock AgentCore Runtime](#), you can split production traffic between foundation and customized models during A/B testing, and [Amazon Bedrock AgentCore Evaluations](#) runs LLM-as-a-Judge assessments against both arms. Accept quantization only when accuracy loss stays within your acceptable quality threshold on task success rate. Treat customization as a pipeline: periodically extract high-quality examples from production logs, schedule quarterly refresh jobs, and gate promotion on A/B validation so drift doesn't compound silently between refreshes.

Implementation steps

1. **Conduct a customization cost-benefit analysis:** Calculate current monthly inference costs for high-volume task categories, identify where training costs amortize within your planning horizon, and compare fine-tuning investment (training compute plus ongoing maintenance) against projected cumulative inference savings.
2. **Curate training data from production logs:** Extract high-quality examples from production invocation logs by filtering for invocations with low error rates and acceptable latency using AgentCore Observability metrics. Target 500 to 1,000 examples per task category. Query Amazon CloudWatch for invocations where latency falls within the p50 to p90 range and error_type is absent, review a sample manually to verify quality, and store the curated dataset in Amazon S3.
3. **Run distillation or fine-tuning:** Use [Amazon Bedrock model customization](#) jobs or [Amazon SageMaker AI AI Training Jobs](#) with QLoRA, and validate using [Amazon Bedrock AgentCore Evaluations](#) against a held-out test set.
4. **Import and A/B test customized models:** Use [Amazon Bedrock Custom Model Import](#) and deploy through [Amazon Bedrock AgentCore Runtime](#), routing a traffic slice to the customized model before promoting it to handle production volume.
5. **Schedule quarterly refresh jobs:** Automate training data extraction and retraining on a quarterly cadence, with A/B validation as the promotion gate to catch drift at each refresh rather than at annual review.

Resources

Related best practices:

- [AGENTCOST02-BP01 Architect tiered model selection for cost-performance optimization](#)
- [AGENTCOST02-BP02 Cost optimize token consumption through efficient prompt engineering](#)

- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)

Related documents:

- [Model customization in Amazon Bedrock](#)
- [Amazon Bedrock Custom Model Import](#)
- [Amazon Bedrock AgentCore Evaluations](#)
- [Evaluate models in Amazon Bedrock](#)
- [Economics for agentic AI on AWS](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)

Related videos:

- [AWS re:Invent 2024 - Mastering model choice: The 3-step Amazon Bedrock advantage \(AIM391\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Evaluations tutorials](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon SageMaker AI](#)
- [Amazon S3](#)

Agent memory and state cost management

Agents with well-tiered memory architectures maintain reasoning quality while keeping storage and retrieval costs proportional to actual usage rather than total history size. Agent memory systems can become expensive through uncontrolled context accumulation, inefficient retrieval patterns, and persistent state storage. Long-running conversations consume growing context windows, and episodic memory retrieval can trigger vector database queries on every reasoning step.

AGENTCOST03: How do you manage agent memory and state costs efficiently?

Capability intent

- Memory storage is aligned to access frequency, with active context on high-performance stores and historical data on lower-cost tiers.
- Retrieval is selective rather than bulk, so the tokens spent per reasoning step reflect the context that actually matters to the current task.
- Long-running sessions compress and prune history continually, keeping token cost sub-linear as conversation length grows.
- Session state and durable agent learning are persisted through managed lifecycle controls, helping prevent unbounded storage growth without discarding knowledge that improves future reasoning.
- Memory and state costs are observable per agent and per session, with alarms surfacing retrieval patterns and storage growth before they become material.

Maturity levels

These levels summarize what each stage of maturity looks like for agent memory and state cost management as a whole.

Level	Name	What it looks like
1	Initial	Memory is a single undifferentiated layer. Full conversation history is passed into every invocation, and there is no retrieval filtering, compression, or lifecycle policy. Per-session memory and state cost isn't measured, so token growth and storage growth are detected only

Level	Name	What it looks like
		when a bill or context overflow forces the issue.
2	Emerging	Short-term and long-term memory are separated, and a basic rolling summarization keeps the longest sessions from unbounded growth. Retrieval uses Amazon Bedrock Knowledge Bases or equivalent vector search with a fixed top-K. Baseline dashboards expose token usage per session, but tuning is one-time and applied by individual teams.
3	Defined	Tiered memory is the standard across agent teams, with managed services such as Amazon Bedrock AgentCore Memory used for short-term and long-term storage. Retrieval is relevance-scored, chunk sizes are right-sized, and context utilization is monitored against a defined target. Lifecycle policies archive or purge stale state on Amazon S3 or equivalent, and Amazon CloudWatch dashboards cover most agents.

Level	Name	What it looks like
4	Proactive	Memory compression, pruning, and selective retrieval are automated defaults, and session lifecycle parameters are tuned per workload. Amazon Bedrock AgentCore Observability provides per-agent and per-session cost attribution, with alarms for context overflow, retrieval hot spots, and unexpected storage growth. Vector index parameters and top-K are reviewed regularly against recall and cost data.
5	Optimized	Memory architecture is continuously optimized from observed access patterns. Top-K, chunk size, tier placement, and checkpoint frequency adapt to workload signals rather than static configuration. Cost regressions on memory and state are caught in pre-production by automated checks against historical baselines, and insights from operations feed back into reasoning pattern and retrieval design.

Common issues to watch for

- Memory is treated as a single undifferentiated store, so every byte pays high-performance storage rates regardless of whether it is ever read again.
- Full conversation history is injected into every invocation, causing token cost to grow linearly with session length and pushing long sessions toward context overflow.
- Retrieval is coarse and indiscriminate, returning large, low-relevance context when a smaller, relevance-scored result set would preserve reasoning quality at a fraction of the cost.
- Session state and durable memory accumulate without a defined lifecycle, leaving storage to grow indefinitely until a cost review forces reactive cleanup.
- Memory and state cost are invisible at the per-agent and per-session level, so regressions such as expensive retrieval patterns, unexpected state growth, and idle session bloat are detected only after they show up on an invoice.

Best practices

- [AGENTCOST03-BP01 Design cost-effective retrieval systems with tiered memory](#)
- [AGENTCOST03-BP02 Cost optimize through intelligent compression and pruning of context windows](#)
- [AGENTCOST03-BP03 Implement cost-optimized state persistence and lifecycle management](#)

AGENTCOST03-BP01 Design cost-effective retrieval systems with tiered memory

Agent memory has to serve two opposing needs at once: fast access for active context, and cheap storage for history that is rarely touched. Tiered memory matches each class of data to infrastructure priced for its actual access pattern, and selective retrieval keeps token costs proportional to what the current task needs.

Desired outcome:

- You have short-term working memory on high-performance storage and long-term memory on cost-effective tiers, with automatic lifecycle transitions between them.
- You retrieve only top-K relevant items per reasoning step rather than loading full memory stores into context.

- You track retrieval operations per session and use the data to tune tier assignments and access patterns.

Common anti-patterns:

- Storing all agent memory in expensive high-performance storage regardless of access frequency, incurring unnecessary costs for rarely accessed historical interactions.
- Retrieving entire memory stores for each reasoning step, consuming excessive input tokens when targeted top-K retrieval would suffice.
- Using single-tier storage for all memory regardless of access pattern, wasting resources on uniform infrastructure for data with distinct access profiles.
- Deploying memory systems without retrieval cost monitoring, hiding inefficient access patterns inside aggregate session cost.

Benefits of establishing this best practice:

- Tiered storage matches each memory category to its access pattern, reducing costs for historical data without sacrificing active session performance.
- Selective top-K retrieval limits context to the most pertinent items, avoiding token charges for irrelevant historical data.
- Automated tier lifecycle management scales across thousands of sessions without manual intervention or over-provisioning.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The cost of agent memory comes from two decisions: where data lives and how much of it you pull into the model's context window. [Amazon Bedrock AgentCore Memory](#) handles the first decision as a managed service. Short-term memory stores turn-by-turn session context on fast storage, while long-term memory extracts and consolidates key insights across sessions into cheaper tiers.

For agents on [Amazon Bedrock AgentCore Runtime](#), this removes the need to build storage tiers and promotion policies by hand. When a custom implementation is required, define explicit promotion and demotion policies based on access frequency so frequently accessed items stay on low-latency storage and rarely accessed items migrate to lower-cost tiers automatically.

Retrieval volume is the second decision, and it has a direct effect on input token cost. [Amazon Bedrock Knowledge Bases](#) provides managed vector retrieval with semantic search. K (the number of chunks returned per query) is the central cost-quality knob: higher K gives the agent more context but pushes more tokens into every invocation. Start with $K=5$ and tune against the trade-off between completeness and cost, not from a preference for safety.

Index design is a less obvious but still important cost consideration. For [Amazon OpenSearch Service Serverless](#)-backed Knowledge Bases, HNSW parameters (`ef_construction` and `m`) balance index build cost against query accuracy and recall. OpenSearch Serverless charges based on indexed data volume and query compute, so tuning these parameters is a direct cost decision, not just a quality decision. Higher `ef_construction` values improve recall but raise both build and query cost, while lower values reduce cost but risk missing relevant items.

Additionally, consider retrieval batching. Pre-fetching the full task context at initiation and caching it in the agent's working memory avoids per-step retrieval overhead. [Amazon Bedrock AgentCore Observability](#) provides OpenTelemetry-compatible telemetry that identifies which retrieval patterns drive the most token consumption, and Amazon CloudWatch Logs Insights queries reveal access patterns that should inform tier reassignments.

Implementation steps

1. **Adopt managed tiered memory:** Integrate [Amazon Bedrock AgentCore Memory](#) for short-term and long-term memory with automatic lifecycle management, and document which namespaces each agent writes to and reads from.
2. **Configure selective retrieval:** Use [Amazon Bedrock Knowledge Bases](#) with top- K semantic search, starting at $K=5$ and tuning based on observed reasoning quality and token cost.
3. **Tune vector index parameters:** Adjust HNSW `ef_construction` and `m` on the [Amazon OpenSearch Service Serverless](#) backing store to balance index build cost, query latency, and recall accuracy for your workload.
4. **Pre-fetch context at task initiation:** Replace per-step retrievals with a single batch pre-fetch at task start, cached in working context so the model doesn't pay retrieval overhead on every reasoning step.
5. **Instrument retrieval operations:** Enable [Amazon Bedrock AgentCore Observability](#) and set Amazon CloudWatch alarms when retrieval frequency exceeds expected bounds per session.
6. **Review access patterns weekly:** Run CloudWatch Logs Insights queries to reveal expensive retrieval patterns and never-accessed items, and use the results to reassign tiers and retire dead entries.

Resources

Related best practices:

- [AGENTCOST01-BP02 Optimize multi-agent collaboration cost through efficient handoff patterns](#)
- [AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations](#)
- [AGENTCOST03-BP02 Cost optimize through intelligent compression and pruning of context windows](#)
- [AGENTCOST03-BP03 Implement cost-optimized state persistence and lifecycle management](#)

Related documents:

- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock Knowledge Bases](#)
- [Economics for agentic AI on AWS](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Memory](#)
- [AWS 2025 - AgentCore Memory: Episodic Memory & Patterns](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Memory tutorials](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore - Lab 2: Memory](#)
- [Diving Deep into Bedrock AgentCore - Memory](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTCOST03-BP02 Cost optimize through intelligent compression and pruning of context windows

In long-running agent sessions, raw conversation history can silently drive costs up, as every turn gets paid for again on every subsequent invocation. Compression, selective retrieval, and pruning keep context proportional to what the agent needs for the current decision rather than growing with session length.

Desired outcome:

- You compress older conversation turns into summaries so historical context doesn't multiply per-invocation token cost.
- You retrieve only the top-K most relevant memory items per reasoning step.
- You prune duplicates, superseded reasoning, and irrelevant tool results before each invocation.
- You monitor context window utilization and alert on sessions approaching overflow.

Common anti-patterns:

- Including full conversation history in every invocation regardless of task relevance, causing linear token cost growth with session length.
- Allowing raw interaction history to accumulate without compression, so context windows are dominated by historical turns with diminishing value.
- Deploying agents without context utilization monitoring, missing sessions that approach overflow thresholds and trigger costly re-invocation errors.
- Retrieving excessive RAG chunks or oversized chunk lengths when smaller, targeted retrievals would maintain reasoning quality at lower cost.
- Failing to prune duplicate or superseded information, paying tokens on content that doesn't contribute to the current reasoning task.

Benefits of establishing this best practice:

- History compression helps prevent linear token cost growth in long-running sessions, making persistent assistants economically viable.
- Selective retrieval includes only high-value context relevant to the current task, reducing token waste from marginally relevant data.

- Context window monitoring helps prevent overflow errors that trigger costly re-invocation with truncated context.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock AgentCore Memory](#) separates short-term and long-term memory, which is the architectural pattern behind rolling summarization. Short-term memory holds raw recent turns, and long-term memory automatically extracts and consolidates key insights across sessions. For agents on AgentCore Runtime, this dual-tier behavior implements rolling summarization without custom code, and it is the difference between a persistent assistant whose token cost is bounded and one whose cost grows linearly with conversation age.

Selective retrieval helps handle the problem of conversation history cost. AgentCore Memory's RetrieveMemoryRecords operation performs semantic search with relevance scoring and metadata filtering, so you can pre-filter by recency or topic before the similarity search runs. Configure top-K between three and five items per reasoning step.

Context pruning assists with retrieval by removing duplicates between summaries and recent turns before each invocation, dropping superseded reasoning steps, and stripping irrelevant tool results. The goal is a target context utilization of 60 to 80% of the model's window, which leaves enough headroom for responses while still benefiting from available context.

RAG chunk sizing also helps solve this problem. When retrieving from [Amazon Bedrock Knowledge Bases](#), chunk sizes of 256 to 512 tokens balance retrieval precision against context bloat, and limiting retrieved chunks to the minimum needed helps prevent marginally relevant data from crowding out the current task. The verification that compression isn't silently hurting quality is a correlation check: pair context utilization with task success rate in CloudWatch Logs Insights and track whether aggressive pruning correlates with success-rate degradation.

[Amazon Bedrock AgentCore Observability](#) exposes token usage metrics that feed CloudWatch dashboards and alarms. Alarms on sessions consistently above 80% utilization flag the candidates for tighter summarization, correlating those same metrics with task success rates confirms whether the compression is paying off in cost without paying in quality.

Implementation steps

1. **Adopt managed rolling summarization:** Integrate [Amazon Bedrock AgentCore Memory](#) for managed compression, or implement custom rolling summarization that compresses the oldest N turns after every N turns.
2. **Configure relevance-scored retrieval:** Use AgentCore Memory's RetrieveMemoryRecords with relevance thresholds and metadata filtering, retrieving only the top-K most relevant items per reasoning step.
3. **Prune context before each invocation:** Remove duplicates, superseded reasoning steps, and irrelevant tool results before each model call so the context window reflects what the current decision needs.
4. **Tune RAG chunk size:** Optimize [Amazon Bedrock Knowledge Bases](#) chunk sizes to 256 to 512 tokens, limit retrieved chunks to the minimum needed, and add re-ranking to maximize relevance.
5. **Alarm on context utilization:** Build Amazon CloudWatch dashboards for context window utilization and set alarms for sessions exceeding 80% utilization.
6. **Correlate utilization with task success:** Use CloudWatch Logs Insights to correlate context utilization with task success rates, validating that compression strategies reduce cost without degrading reasoning quality.

Resources

Related best practices:

- [AGENTCOST02-BP02 Cost optimize token consumption through efficient prompt engineering](#)
- [AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations](#)
- [AGENTCOST03-BP01 Design cost-effective retrieval systems with tiered memory](#)
- [AGENTCOST03-BP03 Implement cost-optimized state persistence and lifecycle management](#)

Related documents:

- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Economics for agentic AI on AWS](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Memory](#)
- [AWS 2025 - AgentCore Observability: Monitor and Debug with OpenTelemetry](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Memory tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTCOST03-BP03 Implement cost-optimized state persistence and lifecycle management

Agent state grows quickly when every reasoning step triggers a checkpoint, and it stays forever when no lifecycle policy removes it. Saving state at meaningful decision points, tiering by access pattern, and automating archival keeps recoverability without paying for a growing backlog of stale sessions.

Desired outcome:

- You checkpoint at meaningful decision points rather than after every reasoning step.
- You have session state tiered by access pattern, with high-performance storage reserved for active work.
- You have automated lifecycle policies that archive or purge stale context.
- You track storage cost per agent and session, with alarms for unexpected growth.

Common anti-patterns:

- Checkpointing after every reasoning step with synchronous writes when asynchronous checkpoints at meaningful decision points would suffice.
- Keeping all session state on high-performance storage regardless of activity level, paying unnecessary costs for inactive or archived sessions.

- Allowing agent memory to accumulate indefinitely without archival or deletion, producing unbounded storage growth.
- Storing agent state uncompressed when compression could reduce storage costs proportionally.
- Deploying state persistence without cost monitoring, hiding high-cost patterns and optimization opportunities.

Benefits of establishing this best practice:

- Automated lifecycle management helps prevent unbounded storage growth without manual intervention across thousands of sessions.
- Managed memory separates durable learning from ephemeral state, keeping knowledge that improves agent performance while cleaning up temporary artifacts.
- Session timeout configuration balances responsiveness with cost by controlling the compute lifecycle.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

[Amazon Bedrock AgentCore Runtime](#) provides a persistent filesystem that handles tiered storage and lifecycle automatically. The filesystem survives session stop and resume cycles for up to 14 days of inactivity before automatic deletion, and the two lifecycle parameters that shape cost are `idleRuntimeSessionTimeout` (default 15 minutes) and `maxLifetime` (up to 8 hours). The 15-minute default suits interactive workloads, while longer timeouts reduce session state transitions for batch workloads. Session storage automatically synchronizes filesystem writes to durable storage throughout the session lifecycle, with data flushed during graceful shutdown when sessions stop.

Make a deliberate design choice about checkpointing. For use cases that require explicit checkpoints at application-defined decision points, implement custom checkpoint logic that writes state snapshots to the persistent filesystem. Checkpoint interval is a trade-off between recovery granularity and storage consumption: more frequent checkpoints enable finer-grained recovery but increase storage cost. Some agent frameworks provide built-in checkpoint capabilities using the same filesystem, which avoids reinventing the pattern.

Consider how you accomplish durable learning. [Amazon Bedrock AgentCore Memory](#) persists insights across sessions (short-term memory for recent interactions, long-term memory for consolidated learning), which is different from per-session filesystem state. For compliance

retention beyond the 14-day Runtime filesystem window, export completed session data to [Amazon S3](#) with Intelligent-Tiering enabled, and configure lifecycle rules for cost-effective long-term storage. Monitor consumption per agent type using [Amazon Bedrock AgentCore Observability](#) with custom dimensions, and set Amazon CloudWatch alarms when growth exceeds expected bounds.

Implementation steps

1. **Deploy on AgentCore Runtime with tuned lifecycle parameters:** Use [Amazon Bedrock AgentCore Runtime](#) for automatic session lifecycle management, configuring `idleRuntimeSessionTimeout` and `maxLifetime` based on workload patterns.
2. **Integrate managed memory for durable learning:** Configure [Amazon Bedrock AgentCore Memory](#) short-term memory for recent context and long-term memory for persistent insights, keeping durable learning separate from ephemeral filesystem state.
3. **Archive compliance-required sessions to S3:** Export completed session histories to [Amazon S3 Intelligent-Tiering](#) and set lifecycle rules for cost-effective long-term retention.
4. **Monitor storage per agent type:** Configure [Amazon Bedrock AgentCore Observability](#) with custom dimensions to track storage cost per agent type, and set Amazon CloudWatch alarms for unexpected storage growth.

Resources

Related best practices:

- [AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops](#)
- [AGENTCOST03-BP01 Design cost-effective retrieval systems with tiered memory](#)
- [AGENTCOST03-BP02 Cost optimize through intelligent compression and pruning of context windows](#)
- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime Sessions](#)
- [Amazon Bedrock AgentCore Memory](#)
- [Amazon S3 Intelligent-Tiering](#)
- [Economics for agentic AI on AWS](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Runtime](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Runtime advanced concepts](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon S3](#)
- [Amazon CloudWatch](#)

Agent tool serving cost optimization

Agents that evaluate tool necessity before invocation and cache results for recurring patterns keep tool costs predictable even as autonomous capabilities expand. Agent tool usage can create unpredictable cost spikes through excessive API calls, failed invocation retry storms, and always-on infrastructure that scales poorly with demand.

AGENTCOST04: How do you optimize agent tool invocation costs?

Capability intent

- Tools are invoked only when needed, with agents designed to consult context, managed memory, and prior tool results before issuing new calls.
- Tool selection favors cheaper alternatives through cost-ranked rubrics, and tool interfaces accept batched inputs to reduce per-call overhead.
- Tool serving infrastructure is consumption-based and scales to zero when idle, with shared services spreading fixed overhead across agents rather than duplicating it per agent.
- Tool results are cached through a layered strategy. Session-scoped caches serve within-session reuse and distributed caches serve cross-session reuse, including semantic matches of functionally equivalent calls.

- Failures are contained through automatic cutoffs and fallback tools designed to preserve agent functionality without unbounded retry costs.
- Per-agent, per-tool invocation metrics and costs are visible at the session and reasoning-cycle level, enabling targeted optimization.

Maturity levels

These levels summarize what each stage of maturity looks like for agent tool serving cost optimization as a whole.

Level	Name	What it looks like
1	Initial	Tool serving infrastructure runs continuously regardless of agent activity. Tools are invoked without checking whether the required information already exists in context or memory, and retries are effectively unlimited. There is no cost attribution per tool or per agent, and no caching strategy. Failures cascade into retry storms because cutoffs are absent.
2	Emerging	Tool serving moves onto managed, consumption-based infrastructure such as Amazon Bedrock AgentCore Gateway and AgentCore Runtime . Basic session-scoped caching and exponential backoff with caps are implemented. Agents receive a cost-ranked tool selection

Level	Name	What it looks like
		rubric in their system prompts, and invocation telemetry is captured through AgentCore Observability . Cost attribution exists at the service level but not currently at the agent or reasoning-cycle level.
3	Defined	Tool interfaces are designed for batched inputs and complete result sets. Session-scoped caching is complemented by distributed, cross-session caching on Amazon OpenSearch Service Serverless . Automatic cutoffs are enforced through AgentCore Policy rather than application logic, and fallback tools preserve agent functionality during degradation. Per-agent and per-tool cost attribution is reported through AWS Cost Explorer tags, and tool-selection accuracy is evaluated periodically with AgentCore Evaluations .

Level	Name	What it looks like
4	Proactive	Semantic caching recognizes functionally equivalent tool calls through embedding similarity, and cache time to live (TTL) values are calibrated per tool based on data volatility. Input validation in AWS Lambda action group functions helps prevent wasted invocations from malformed calls. Cutoffs and retry budgets are codified in Cedar policies enforced at the Gateway boundary. Real-time dashboards track cache hit rates, cutoff state transitions, and retry cost as a proportion of total tool cost.

Level	Name	What it looks like
5	Optimized	Tool serving is self-optimizing. Cache TTLs, cutoff thresholds, and tool-selection rubrics adjust automatically based on observed hit rates, failure patterns, and cost outcomes. Event-driven cache invalidation purges stale data immediately when source systems change, supporting aggressive caching without staleness. Tool-selection and retry patterns are continuously evaluated against business outcomes, and optimization findings feed back into system prompts, policies, and infrastructure automatically.

Common issues to watch for

- Agents invoke tools without first checking context or managed memory, generating duplicate calls across reasoning iterations and driving up per-session costs.
- Narrow tool interfaces return minimal data, forcing agents to chain follow-up calls to assemble complete context and multiplying per-call overhead.
- Persistent tool servers run continuously for unpredictable agent workloads, incurring charges during long idle windows when no agent is invoking them.
- Retry logic without automatic cutoffs turns transient service degradation into cost-amplifying retry storms that don't resolve the underlying failure.
- Caching strategies rely solely on exact-match lookups and miss functionally equivalent calls that the agent phrases differently across sessions or reasoning iterations.
- Tool invocation telemetry stops at the service level, reducing the risk of per-agent or per-reasoning-cycle attribution and leaving optimization effort untargeted.

Best practices

- [AGENTCOST04-BP01 Design cost effective tool selection to minimize unnecessary invocations](#)
- [AGENTCOST04-BP02 Cost optimize tool serving through serverless and resource sharing](#)
- [AGENTCOST04-BP03 Implement intelligent caching and failure handling for tool results](#)

AGENTCOST04-BP01 Design cost effective tool selection to minimize unnecessary invocations

Often, the most cost-effective tool call is the one an agent decides not to make because the answer is already in context. Context-first reasoning, cost-ranked selection, and duplicate detection tie tool invocation to the value of the information retrieved.

Desired outcome:

- You have agents checking context and managed memory before invoking tools.
- You have a cost-ranked selection rubric that points agents to cheaper alternatives first.
- You batch requests where possible and cache results within sessions to avoid duplicate calls.
- You monitor per-tool invocation frequency and cache hit rates as distinct metrics.

Common anti-patterns:

- Invoking tools without checking whether required information already exists in context or managed memory, adding cost without improving results.
- Creating narrow tool interfaces that return minimal data, forcing follow-up calls to assemble complete context.
- Implementing retry logic without exponential backoff or automatic cutoffs, causing retry storms that multiply costs during service degradation.
- Operating without tool invocation metrics, so no one can identify which tools are most expensive or most frequently called.

Benefits of establishing this best practice:

- Context-first evaluation reduces unnecessary tool invocations for agents with rich context from prior reasoning steps.

- Cost-ranked tool selection rubrics direct agents to cheaper alternatives, reserving expensive external APIs for cases where lower-cost options are insufficient.
- Batched tool interfaces and complete result sets reduce per-call overhead and the need for follow-up invocations.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Tool necessity belongs in the agent's reasoning prompt, not as an afterthought in monitoring. The system prompt should instruct the model to assess whether the answer can be derived from information already in context, from conversation history stored in [Amazon Bedrock AgentCore Memory](#), or from prior tool results within the same reasoning cycle, before selecting a tool. Pair this with a cost-ranked selection rubric that places cheaper alternatives first: if a local computation or a cached result produces the same answer as an external API call, the agent should take the cheaper path.

Tool interface design matters as much as agent instructions. Narrow interfaces that return minimal data force agents to make follow-up calls to assemble the context they need, which inflates per-reasoning-cycle tool cost. [Amazon Bedrock AgentCore Gateway](#) provides MCP-based tool discovery with composition features that combine multiple APIs into single endpoints, reducing invocation overhead. Design tool interfaces to accept batch inputs and return complete result sets so a single call does the work of many.

Consider how you implement duplicate detection. Agents often invoke the same tool with identical parameters across reasoning iterations, especially when revisiting a branch. Implement a session-scoped tool result cache in your action group Lambda functions or AgentCore Gateway MCP servers so the agent doesn't re-invoke the same tool with the same parameters. Store results in AgentCore Memory's short-term memory so the agent's reasoning prompt can reveal prior results before deciding to make another call.

For enforcement and measurement, [Amazon Bedrock AgentCore Policy](#) applies Cedar policies that halt retries when failure rates indicate persistent degradation and cap tool calls per reasoning cycle. [Amazon Bedrock AgentCore Observability](#) exposes tool selection patterns, Amazon CloudWatch tracks invocation frequency and deduplication hit rates, and [Amazon Bedrock AgentCore Evaluations](#) scores tool selection accuracy so patterns of over-invocation appear as quality data, not just cost data.

Implementation steps

1. **Embed tool necessity evaluation in system prompts:** Direct the model to check context and [Amazon Bedrock AgentCore Memory](#) before invoking tools, and include a cost-ranked selection rubric that places cheaper alternatives first.
2. **Redesign tool interfaces for batching:** Expose tools through [Amazon Bedrock AgentCore Gateway](#) with batch inputs and complete result sets so one call carries the payload that previously required several.
3. **Cache tool results within sessions:** Implement session-scoped caches in action group Lambda functions or Gateway MCP servers to deduplicate identical tool calls, storing results in AgentCore Memory so the agent can surface them before the next call.
4. **Apply automatic cutoffs through policy:** Configure [Amazon Bedrock AgentCore Policy](#) Cedar policies that cap tool calls per reasoning cycle and halt retries on persistent failures.
5. **Monitor tool selection patterns:** Use [Amazon Bedrock AgentCore Observability](#) to surface tool selection patterns and create Amazon CloudWatch metrics for tool invocation frequency and deduplication hit rates.
6. **Score tool selection accuracy:** Use [Amazon Bedrock AgentCore Evaluations](#) to periodically score tool selection accuracy, flagging patterns where agents choose expensive tools when cheaper alternatives would suffice.

Resources

Related best practices:

- [AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops](#)
- [AGENTCOST02-BP02 Cost optimize token consumption through efficient prompt engineering](#)
- [AGENTCOST04-BP02 Cost optimize tool serving through serverless and resource sharing](#)
- [AGENTCOST04-BP03 Implement intelligent caching and failure handling for tool results](#)
- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock AgentCore Memory](#)

- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Gateway](#)
- [AWS re:Invent 2024 - Scale agent tools with AgentCore Gateway \(AIM3313\)](#)
- [Integrating MCP Tools with Strands Agents](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Gateway tutorials](#)

Related workshops:

- [Diving Deep into Bedrock AgentCore - Gateway](#)

Related tools:

- [Strands Agents MCP Tools](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

AGENTCOST04-BP02 Cost optimize tool serving through serverless and resource sharing

Tool infrastructure that runs constantly to serve unpredictable agent traffic carries the highest idle cost in an agent stack. Serverless tool serving with shared infrastructure across agents aligns spend with actual invocations and removes the fixed overhead of per-agent dedicated instances.

Desired outcome:

- You have tool-serving infrastructure that scales dynamically with agent usage and charges only for actual invocations.

- You share stateless tool services across agents while maintaining security isolation.
- You use private networking and compact serialization to reduce data transfer costs on high-frequency tool invocations.
- You track per-agent cost attribution for targeted optimization.

Common anti-patterns:

- Running persistent servers for tool serving that incur charges during hours or days when no agents invoke tools.
- Creating dedicated tool server instances per agent rather than shared stateless services, producing dozens of underutilized servers.
- Routing tool invocations through NAT Gateways when agents and tools live in the same VPC, incurring unnecessary per-GB data processing charges.

Benefits of establishing this best practice:

- Serverless tool serving scales to zero when agents are inactive, reducing idle costs through consumption-based pricing.
- Shared tool infrastructure spreads fixed hosting overhead across all agents while maintaining security isolation.
- VPC endpoints and compact serialization reduce data transfer costs for high-frequency tool invocations.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Agent tool traffic is inherently bursty. An agent fleet is active during business hours, idle overnight, and heavily uneven across agent types. Provisioned tool infrastructure pays for that shape by keeping compute warm through idle hours, which can be a significant hidden cost in agent stacks. [Amazon Bedrock AgentCore Gateway](#) provides fully-managed, serverless tool serving that converts APIs and existing services into MCP-compatible tools without infrastructure management. AgentCore Gateway handles authentication, scales automatically, and combines multiple APIs into unified endpoints.

Because tools exposed through AgentCore Gateway are available to all authorized agents, one endpoint can serve an entire fleet rather than one endpoint per agent. [Amazon Bedrock AgentCore Identity](#) and [Amazon Bedrock AgentCore Policy](#) supply the fine-grained access control that keeps sharing safe.

For tools that need extended execution, [Amazon Bedrock AgentCore Runtime](#) supports workloads up to 8 hours with consumption-based pricing calculated at per-second increments. Consumption pricing is the right default for unpredictable tool invocation patterns because it charges only during active processing.

Cold starts are a failure mode worth planning for. A cold tool extends the agent's reasoning cycle and may trigger retries, which can push per-session token costs up on cold paths. Monitor cold start frequency in Amazon CloudWatch and evaluate Lambda SnapStart or scheduled warming when cold starts are material.

Networking and serialization are the foundation of planning for these failure modes. VPC endpoints for private data paths avoid NAT Gateway processing charges for high-frequency tool invocations between agents and tools in the same VPC. Compact JSON (or binary formats where supported) reduces payload sizes on repeated high-frequency calls. Tagging every invocation with agent ID and workflow ID lets [Amazon Bedrock AgentCore Observability](#) and AWS Cost Explorer reveal which agents and tools drive the highest spend.

Implementation steps

1. **Expose tools through serverless Gateway:** Deploy agent tools through [Amazon Bedrock AgentCore Gateway](#) MCP server capabilities for serverless infrastructure with automatic scaling and shared access across agents.
2. **Apply fine-grained access control:** Configure [Amazon Bedrock AgentCore Policy](#) Cedar policies for per-agent tool access, preserving security isolation while sharing infrastructure.
3. **Attribute cost per agent:** Use [Amazon Bedrock AgentCore Observability](#) telemetry tags with agent ID and workflow ID, and generate periodic AWS Cost Explorer reports by agent and tool type.
4. **Monitor invocation patterns and cold starts:** Expose tool invocation patterns through AgentCore Observability, and set Amazon CloudWatch alarms for patterns that exceed expected bounds, including cold start frequency.

Resources

Related best practices:

- [AGENTCOST04-BP01 Design cost effective tool selection to minimize unnecessary invocations](#)
- [AGENTCOST04-BP03 Implement intelligent caching and failure handling for tool results](#)
- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Gateway](#)
- [AWS 2025 - AgentCore Deep Dive: Runtime](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Gateway tutorials](#)

Related workshops:

- [Getting started with Amazon Bedrock AgentCore - Lab 3: Gateway, Identity & Policy](#)
- [Diving Deep into Bedrock AgentCore - Gateway](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [AWS Cost Explorer](#)

AGENTCOST04-BP03 Implement intelligent caching and failure handling for tool results

Tool costs can be unpredictable when agents repeat identical or equivalent calls, and they can spike sharply when retries run unbounded through a service outage. Two-layer caching, schema validation, and automatic cutoffs convert those failure modes into predictable, bounded costs.

Desired outcome:

- You have session-scoped and cross-session semantic caches reducing redundant tool invocations.
- You validate tool inputs against JSON Schema before invocation to help prevent wasted calls on malformed requests.
- You have automatic cutoffs that halt retries when failure rates exceed thresholds, with automatic fallback to alternative tools.
- You track cache hit rates and retry costs as distinct metrics.

Common anti-patterns:

- Not caching frequently used tool results, making repeated identical calls within the same session that waste compute and external API costs.
- Using only exact-match caching when agents phrase the same request differently, missing cache hits for semantically identical calls.
- Retrying failed tool invocations indefinitely without automatic cutoffs, multiplying cost during service degradation without resolving the underlying issue.
- Not validating tool input schemas before invocation, allowing malformed calls to waste invocation cost without producing usable results.

Benefits of establishing this best practice:

- Two-layer caching reduces redundant tool invocations and external API charges.
- Automatic cutoffs halt retries when failure rates exceed thresholds, helping prevent expensive retry storms.
- Event-driven cache invalidation supports aggressive caching of volatile data by purging stale results promptly when source data changes.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Tool caching has to work at two scopes to cover both obvious and non-obvious repetition. The session-scoped layer works through [Amazon Bedrock AgentCore Runtime](#) and catches duplicate calls within a single agent session, which is a common failure mode when agents revisit a reasoning branch. The cross-session layer uses [Amazon OpenSearch Service Serverless](#) for semantic caching: generate embeddings of tool parameters and query for similar prior calls above a cosine similarity threshold before invoking the tool. Each cache entry's TTL should be calibrated to the underlying data's volatility. For example, a weather API's freshness requirement is minutes, while a static reference knowledge base tolerates hours or days.

Schema validation can help prevent waste. Agents sometimes generate tool calls with incorrect parameter types, missing required fields, or invalid enum values, and those calls pay tool-serving and external API costs for a response that can't be used. JSON schema validation in the action group Lambda function rejects malformed requests before they reach external APIs and returns a validation error to the agent for correction.

Cache invalidation can help make aggressive caching safer. Event-driven invalidation listens for source-data changes and purges affected cache entries immediately, so volatile data can still be cached without returning stale results. Without event-driven invalidation, teams end up choosing between aggressive TTLs (stale results) or short TTLs (low hit rates), and both options leave cost on the table.

For failure handling, [Amazon Bedrock AgentCore Policy](#) Cedar policies enforce automatic cutoffs when failure rates exceed thresholds, halting retry storms during service degradation. Automatic fallback to alternative tools maintains agent functionality during outages, and retry budgets per reasoning session cap total retry attempts using exponential backoff with jitter. Cache and retry telemetry is exposed through [Amazon Bedrock AgentCore Observability](#) and Amazon CloudWatch: hit rates per layer, cutoff state transitions, and retry cost as a percentage of total tool cost. For caching that extends beyond tool results into model invocations, see [AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations](#).

Implementation steps

1. **Deploy two-layer caching:** Implement a session-scoped in-process cache on [Amazon Bedrock AgentCore Runtime](#) and an [Amazon OpenSearch Service Serverless](#) semantic cache for cross-

- session reuse, with TTLs calibrated per tool (short for volatile data, long for static reference data).
2. **Deploy semantic caching:** Generate parameter embeddings and query OpenSearch Serverless for similar prior calls above a cosine similarity threshold before invoking the tool.
 3. **Validate tool inputs:** Implement JSON Schema validation in action group Lambda functions to reject malformed requests before they reach external APIs, returning validation errors for the agent to correct.
 4. **Enforce cutoffs and fallback tools:** Configure [Amazon Bedrock AgentCore Policy](#) Cedar policies for automatic cutoffs, wire automatic fallback to alternative tools when cutoffs activate, and set retry budgets per reasoning session.
 5. **Monitor cache and retry metrics:** Create Amazon CloudWatch metrics for cache hit rates, cutoff transitions, and retry costs using [Amazon Bedrock AgentCore Observability](#), with alarms for degraded performance.

Resources

Related best practices:

- [AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations](#)
- [AGENTCOST04-BP01 Design cost effective tool selection to minimize unnecessary invocations](#)
- [AGENTCOST04-BP02 Cost optimize tool serving through serverless and resource sharing](#)
- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)

Related documents:

- [Amazon Bedrock AgentCore Policy](#)
- [Optimize LLM response costs and latency with effective caching](#)
- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Economics for agentic AI on AWS](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Runtime](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Gateway tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon OpenSearch Service](#)
- [Amazon CloudWatch](#)

Agent cost visibility and attribution

Organizations that instrument agent costs at the reasoning-cycle level make data-driven optimization decisions that compound savings across their entire agent fleet. Traditional cost tracking fails with agentic systems because costs span reasoning cycles, multi-agent workflows, tool invocations, and tenant boundaries that standard billing can't capture.

AGENTCOST05: How do you implement cost attribution?

Capability intent

- Agent spending is attributable at the reasoning-cycle, agent, workflow, and tenant level rather than only at the account level.
- Orchestration overhead and worker execution costs are separable, so teams can tell where each dollar is spent.
- Cost traces follow every handoff in multi-agent workflows, producing a true cost-per-workflow-completion.
- Tenant identifiers flow through all agent operations, supporting consumption-based billing, noisy-neighbor detection, and quota enforcement.
- Technical telemetry is translated into business metrics. Cost-per-decision, cost-per-task-completion, and ROI against manual processes are reported, and they are available to business stakeholders through self-service dashboards.

Maturity levels

These levels summarize what each stage of maturity looks like for agent cost visibility and attribution as a whole.

Level	Name	What it looks like
1	Initial	Agent costs are visible only at the AWS account level. No standard tag taxonomy exists, and AWS Cost Explorer reports can't separate agents, workflows, or tenants. Teams react to billing surprises after the fact because per-agent and per-reasoning-phase attribution is missing.
2	Emerging	A standard tag taxonomy (agent-id, agent-role, workflow-id, task-type, environment) is applied to Amazon Bedrock invocations and Amazon Bedrock AgentCore Runtime sessions. Amazon Bedrock AgentCore Observability exports traces to Amazon CloudWatch , and tag-based cost allocation is activated in AWS Cost Explorer. Per-agent and per-workflow reports exist, but coverage is uneven and multitenant attribution is limited.

Level	Name	What it looks like
3	Defined	Workflow trace IDs propagate through every agent invocation, tool call, and memory operation. Orchestration overhead ratios, cost-per-reasoning-cycle, and cost-per-task-completion are tracked on CloudWatch dashboard s. AWS Budgets alerts and CloudWatch alarms drive timely intervention. Tenant identifiers flow through Amazon Bedrock AgentCore Identity for multitenant deployments.

Level	Name	What it looks like
4	Proactive	Cost aggregation rolls up invocation, agent, workflow, and tenant costs automatically. Tenant-level quotas are enforced through Amazon Bedrock AgentCore Policy with Cedar policies , and noisy-neighbor alarms fire when consumption exceeds three times the historical baseline. Business-facing dashboards in Amazon Quick combine AWS Cost and Usage Reports and observability telemetry to show ROI against manual processes. Experimentation infrastructure routes traffic to alternative collaboration patterns so architecture decisions are made with cost-quality data.

Level	Name	What it looks like
5	Optimized	Cost attribution is continuously refined and shapes the organization's agent design decisions. Automated narrative generation from a small Amazon Bedrock model explains cost drivers and recommends optimizations with quantified savings estimates. Pooled-compared-to-dedicated break-even analysis informs tenant placement, per-tenant pricing models adapt to consumption patterns, and business units review cost narratives on a fixed monthly cadence. Cost visibility is a core input to quarterly investment decisions rather than an after-the-fact report.

Common issues to watch for

- Cost reporting stops at the AWS account level, so teams can't separate supervisor overhead from worker execution or identify which reasoning phase consumes disproportionate tokens.
- Tag taxonomies exist on paper but are applied inconsistently across model invocations, function executions, and data operations, which silently breaks per-agent and per-workflow attribution.
- Multi-agent workflows are analyzed one agent at a time because no workflow trace ID propagates across handoffs, producing inflated orchestration costs that go unnoticed.
- Multitenant deployments lack tenant context in metric dimensions and cost allocation tags, which helps prevent consumption-based billing and allows noisy-neighbor tenants to drive infrastructure scaling for everyone.

- Cost data stays inside engineering dashboards, so business stakeholders see only raw token counts and Lambda execution times, and can't translate agent spending into business outcomes or ROI against manual processes.

Best practices

- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)
- [AGENTCOST05-BP02 Implement distributed cost tracing for multi-agent workflows](#)
- [AGENTCOST05-BP03 Design tenant-aware cost allocation for agent as a service \(AaaS\) pricing models](#)
- [AGENTCOST05-BP04 Create chargeback and ROI reporting](#)

AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution

Account-level billing shows what an agent fleet costs, but it doesn't show where the specific costs come from. Granular tracking by agent, workflow, and reasoning phase provides trackable detail about your spending, which makes opaque costs into the input for targeted optimization.

Desired outcome:

- You have a standard tag taxonomy applied consistently across all agent invocations.
- You track per-phase token consumption (planning, execution, reflection, and verification) separately.
- You monitor tool invocation costs separately from model inference costs.
- You calculate cost-per-decision, cost-per-reasoning-cycle, and cost-per-task-completion as primary agent metrics.

Common anti-patterns:

- Tracking only account-level AWS billing without per-agent or per-workflow attribution, reducing the risk of identification of cost drivers.
- Deploying agents without consistent resource tagging across model invocations, function executions, and data operations.
- Monitoring total agent costs without distinguishing between supervisor overhead, worker execution, and individual reasoning phases.

- Monitoring only infrastructure costs without calculating cost-per-autonomous-task-completion, reducing the risk of economic evaluation of agent efficiency.

Benefits of establishing this best practice:

- Agent-specific metrics identify cost anomalies and enable comparison of agent performance across the fleet.
- Per-phase token tracking reveals which reasoning phases consume disproportionate tokens, enabling targeted optimization.
- Business-relevant metrics like cost-per-task-completion enable economic evaluation of different reasoning strategies.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The foundation of agent cost visibility is a tag taxonomy applied consistently everywhere spend happens, like:

- agent-id
- agent-role (like supervisor, worker, and specialist)
- workflow-id
- task-type
- Environment on every [Amazon Bedrock](#) invocation and [Amazon Bedrock AgentCore Runtime](#) session

These tags are the primary key for all downstream cost attribution. Activating tag-based cost allocation in AWS Cost Explorer generates per-agent and per-workflow reports without custom pipeline work, so as soon as tagging is consistent, the reports become usable.

[Amazon Bedrock AgentCore Observability](#) decomposes agent execution into individual operations with token counts and latency through distributed tracing. Per-phase tracking becomes possible because you can attribute tokens to planning, execution, reflection, and verification without manual instrumentation. The AgentCore Runtime consumption-based pricing and microVM session isolation keep cost boundaries aligned with execution boundaries, so the telemetry and the billing see the same unit of work.

A single user request triggers multiple agents, each making multiple model calls, tool invocations, and memory operations, so raw invocation cost has to roll up through a hierarchy to be useful. The aggregation pattern is:

1. Collect per-invocation costs from Amazon Bedrock API responses
2. Associate them with the parent agent using session tags
3. Roll agent costs into workflow totals using workflow-id
4. Attribute workflow costs to tenants for multitenant deployments

Once aggregation is in place, cost reports work at every level: invocation-level for optimization, agent-level for performance comparison, workflow-level for business justification, and tenant-level for billing.

Publishing per-phase token counts as Amazon CloudWatch custom metrics enables you to build dashboards for cost-per-decision, cost-per-reasoning-cycle, and cost-per-task-completion segmented by agent type. CloudWatch alarms on cost-per-task-completion thresholds and AWS Budgets alerts for per-agent monthly spending limits turn the tracking from a passive report into an active signal that tells the team when an agent's economics have shifted.

Implementation steps

1. **Define and apply a standard tag taxonomy:** Apply agent-id, agent-role, workflow-id, task-type, and environment tags consistently to all [Amazon Bedrock](#) invocations and [Amazon Bedrock AgentCore Runtime](#) sessions, and enable AWS Cost Explorer tag-based cost allocation.
2. **Enable end-to-end cost traces:** Configure [Amazon Bedrock AgentCore Observability](#) to capture distributed traces, and export telemetry to Amazon CloudWatch for per-operation cost analysis.
3. **Aggregate costs hierarchically:** Implement a Lambda function that runs every 15 minutes to collect model inference, tool invocation, and memory costs, storing aggregated results by agent and session in a DynamoDB cost tracking table with rollups from invocation to agent to workflow to tenant.
4. **Build cost-per-task dashboards:** Create CloudWatch dashboards displaying cost-per-decision, cost-per-reasoning-cycle, and cost-per-task-completion by agent type.
5. **Configure alerts and budgets:** Set AWS Budgets alerts for per-agent monthly spending limits and CloudWatch alarms for cost-per-task-completion thresholds.

Resources

Related best practices:

- [AGENTCOST05-BP02 Implement distributed cost tracing for multi-agent workflows](#)
- [AGENTCOST05-BP03 Design tenant-aware cost allocation for AaaS pricing models](#)
- [AGENTCOST05-BP04 Create chargeback and ROI reporting](#)
- [AGENTCOST07-BP02 Establish proactive anomaly detection for agent cost patterns](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Using cost allocation tags](#)
- [Economics for agentic AI on AWS](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)

Related videos:

- [AWS 2025 - AgentCore Observability: Monitor and Debug with OpenTelemetry](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Observability tutorials](#)

Related workshops:

- [Diving Deep into Bedrock AgentCore - Observability](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [AWS Cost Explorer](#)
- [AWS Budgets](#)

AGENTCOST05-BP02 Implement distributed cost tracing for multi-agent workflows

In multi-agent workflows, aggregate cost tells you how much you spent but not where those specific costs came from. Tracing with workflow IDs propagating across agent boundaries makes workflow costs more clear, as it is broken down into worker execution, tool invocations, and memory operations, which helps you make data-driven architectural optimization decisions.

Desired outcome:

- You propagate workflow trace IDs through every agent invocation, tool call, and memory operation.
- You calculate true cost-per-workflow-completion, with orchestration overhead tracked separately from execution cost.
- You compare efficiency across different collaboration patterns using real cost data.
- You visualize workflow cost by pattern, agent role, and business outcome.

Common anti-patterns:

- Tracking costs for individual agents without workflow-level correlation, making it impossible to calculate true cost-per-workflow-completion.
- Combining supervisor and worker costs into a single metric, obscuring whether workflows suffer from excessive orchestration overhead.
- Deploying one multi-agent pattern without measuring cost differences between alternatives, missing architectural cost reduction.
- Analyzing total costs without role-based breakdowns, reducing the risk of identification of which agent types drive the highest spending and require targeted optimization.

Benefits of establishing this best practice:

- Full workflow cost visibility enables calculation of true cost-per-workflow-completion across agent boundaries.
- Orchestration overhead ratios reveal when coordination consumes a disproportionate share of workflow spending.

- Cost comparison across collaboration patterns turns architecture decisions from guesswork into data-driven choices.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Workflow-level cost visibility starts with a workflow trace ID generated at workflow initiation and propagated through every agent invocation, tool call, and memory operation. Without that correlation key, per-agent costs can't be stitched into a per-workflow total, and it becomes difficult to determine how expensive it is to deliver one business outcome. [Amazon Bedrock AgentCore Observability](#) provides a three-tiered hierarchy that maps directly to this problem: sessions for complete workflows, traces for individual agent invocations, and spans for operation-level granularity. For agents on [Amazon Bedrock AgentCore Runtime](#), session isolation keeps cost boundaries aligned with agent execution boundaries, so no complex allocation formulas are required.

The most useful decomposition within a workflow is the orchestrator-compared-to-worker split. Tag supervisor invocations with agent-role:orchestrator and worker invocations with agent-role:worker, then compute the orchestration overhead ratio as orchestrator cost divided by total workflow cost. A high ratio indicates coordination is dominating execution, which typically signals a hierarchy that needs flattening or manifests that need compression. Breaking workflow cost further into orchestration tokens, worker execution tokens, tool invocation costs, and memory retrieval costs tells you which component to optimize first.

Cost data alone can be misleading without quality data to correlate. [Amazon Bedrock AgentCore Evaluations](#) measures output quality alongside cost, which turns optimization into a trade-off decision rather than a single-axis minimization. A cheaper workflow pattern that degrades quality isn't actually cheaper in business terms, and the evaluation overlay makes that trade-off explicit.

Routing a percentage of executions to alternative collaboration patterns and comparing cost-per-workflow-completion across patterns in Amazon CloudWatch dashboards and AWS Cost Explorer lets teams pick architectures based on real behavior, not theoretical cost models. For patterns that specifically optimize supervisor costs, see [AGENTCOST01-BP03 Implement cost-effective patterns like hybrid supervisor for multi-agent coordination](#).

Implementation steps

1. **Enable distributed tracing across agents:** Configure [Amazon Bedrock AgentCore Observability](#) to capture distributed traces, exporting telemetry to Amazon CloudWatch with workflow trace IDs propagated through every invocation.
2. **Apply role-based tagging and compute overhead ratios:** Tag every invocation with agent-role (orchestrator or worker), and calculate the orchestration overhead ratio per workflow type.
3. **Visualize cost by pattern and alarm on thresholds:** Build CloudWatch dashboards showing workflow cost distributions by collaboration pattern and agent role, with alarms when orchestration overhead exceeds thresholds.
4. **Run pattern experiments:** Route a percentage of executions to alternative collaboration patterns and compare cost-per-workflow-completion across patterns.
5. **Compare workflow efficiency in Cost Explorer:** Use AWS Cost Explorer to compare efficiency across different collaboration patterns over time.
6. **Decompose workflow cost:** Deploy cost aggregation functions that break total workflow cost into orchestration tokens, worker execution tokens, tool invocation costs, and memory retrieval costs for each workflow type.

Resources

Related best practices:

- [AGENTCOST01-BP02 Optimize multi-agent collaboration cost through efficient handoff patterns](#)
- [AGENTCOST01-BP03 Implement cost-effective patterns like hybrid supervisor for multi-agent coordination](#)
- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)
- [AGENTCOST05-BP03 Design tenant-aware cost allocation for AaaS pricing models](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock AgentCore Runtime](#)
- [Economics for agentic AI on AWS](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)

Related videos:

- [AWS 2025 - AgentCore Observability: Monitor and Debug with OpenTelemetry](#)
- [AWS re:Invent 2024 - Balance cost, performance & reliability for AI at enterprise scale \(AIM3304\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Observability tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [AWS Cost Explorer](#)

AGENTCOST05-BP03 Design tenant-aware cost allocation for agent as a service (AaaS) pricing models

Agent as a service (AaaS) offerings without per-tenant cost attribution can bill only by capacity estimates, can't detect noisy neighbors until infrastructure has already scaled, and can't decide when a tenant should move to dedicated infrastructure. Propagating tenant context through every operation and tracking cost at the tenant level fixes all three problems at once.

Desired outcome:

- You propagate tenant identifiers through all agent operations: model invocations, tool executions, memory operations, and data storage.
- You have flexible cost allocation models supporting per-decision, per-task, and per-agent-hour pricing.
- You detect noisy neighbors before they drive infrastructure scaling that affects all customers.
- You enforce tenant-level budget controls that cap per-tenant spending.

Common anti-patterns:

- Tracking agent costs only at the account level without tenant context, making it impossible to generate accurate tenant invoices.

- Allowing high-usage tenants to consume shared resources without detection, driving infrastructure scaling costs that affect all customers.
- Implementing only one billing approach because the cost allocation system can't support multiple pricing dimensions.
- Allowing unbounded agent costs without tenant-level usage limits, creating financial risk when unexpected usage spikes occur.
- Building agent as a service offerings with a single fixed pricing model, reducing the risk of revenue optimization based on actual usage patterns.

Benefits of establishing this best practice:

- Per-tenant cost tracking enables billing based on actual resource consumption rather than capacity-based estimates.
- Noisy neighbor detection and tenant-level throttling help prevent unexpected infrastructure scaling from aggressive usage patterns.
- Cost allocation data enables data-driven decisions about when dedicated infrastructure becomes more cost-effective than pooled resources.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Tenant context has to travel with every billable event. [Amazon Bedrock AgentCore Identity](#) tags workload identities, credential providers, and API key providers with tenant metadata that propagates through downstream operations. Those tags integrate with AWS Cost Explorer for per-tenant cost breakdowns without requiring separate AWS accounts per tenant. For agents on [Amazon Bedrock AgentCore Runtime](#), session-based architecture provides natural tenant boundaries, and consumption-based pricing means each session's bill reflects actual work done for that tenant.

[Amazon Bedrock AgentCore Observability](#) captures all billable events with tenant identifiers in metric dimensions: token consumption, tool invocation costs from [Amazon Bedrock AgentCore Gateway](#), and memory operation costs from [Amazon Bedrock AgentCore Memory](#). Tenant-level quota enforcement lives in [Amazon Bedrock AgentCore Policy](#) with Cedar policies, which is how you cap per-tenant spending without pushing the enforcement into application code where it can be bypassed.

A *noisy neighbor* is a virtual machine or container that consumes disproportionate system resources. Noisy neighbor detection needs a baseline and a deviation threshold that reflect how agent reasoning depth actually varies. Some tenants execute simple single-step decisions, while others trigger complex multi-turn reasoning chains. An Amazon CloudWatch alarm when a tenant's consumption exceeds three times their historical baseline catches abnormal usage early enough to help prevent infrastructure scaling that increases costs for every customer. The threshold is tuned per workload to avoid under- or over-firing.

Resource-sharing decisions need a cost model. Pooled infrastructure achieves higher utilization and lower per-tenant costs but requires strong isolation. Dedicated infrastructure provides stronger isolation and predictable performance at higher fixed costs. Build a per-tenant break-even model that calculates when a tenant's usage would be cheaper on dedicated infrastructure than on their proportional share of pooled resources, and use it to offer dedicated deployments to tenants who have crossed that line.

Implementation steps

1. **Tag identities for tenant attribution:** Configure [Amazon Bedrock AgentCore Identity](#) with tenant-specific tags on workload identities, and activate the tenant-id cost allocation tag in the AWS billing console.
2. **Deploy agents with session-level tenant tags:** Run agents on [Amazon Bedrock AgentCore Runtime](#) with session-level tenant tagging so cost attribution follows the session, not the pool.
3. **Capture all cost dimensions per tenant:** Configure [Amazon Bedrock AgentCore Observability](#) to export telemetry to Amazon CloudWatch with tenant identifiers in metric dimensions.
4. **Enforce tenant quotas and noisy neighbor detection:** Implement tenant-level quota enforcement through [Amazon Bedrock AgentCore Policy](#), and set CloudWatch alarms for consumption spikes exceeding three times the tenant's historical baseline.
5. **Build a flexible pricing engine:** Support per-decision, per-task, and per-agent-hour billing models with tenant-specific configurations so pricing can evolve without re-architecting the billing pipeline.
6. **Model pooled compared to dedicated break-even:** Use AWS Cost Explorer API data to calculate the break-even point per tenant, identifying tenants approaching the threshold where dedicated AgentCore deployments become cost-effective.

Resources

Related best practices:

- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)
- [AGENTCOST05-BP02 Implement distributed cost tracing for multi-agent workflows](#)
- [AGENTCOST05-BP04 Create chargeback and ROI reporting](#)

Related documents:

- [Amazon Bedrock AgentCore Identity](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Manage multi-tenant Amazon Bedrock costs using application inference profiles](#)
- [AWS SaaS Lens](#)
- [Using cost allocation tags](#)

Related videos:

- [AWS re:Invent 2024 - Building multi-tenant SaaS agents with AgentCore \(SAS407\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Observability tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Cost Explorer](#)

AGENTCOST05-BP04 Create chargeback and ROI reporting

Raw token counts and execution durations are the wrong unit of measure for business stakeholders deciding whether to fund agent capabilities. Translating technical cost into business metrics like cost-per-customer-interaction and comparing against the manual processes agents replace turns agent economics into something non-technical leaders can evaluate against familiar frameworks.

Desired outcome:

- You translate technical agent costs into business metrics through automated chargeback reports.

- You demonstrate agent ROI by comparing agent costs against the manual processes they replace.
- You allocate cost by business unit to create accountability.
- You provide self-service cost dashboards so business teams can act without engineering bottlenecks.

Common anti-patterns:

- Providing only raw technical cost data (like token counts or Lambda execution times) without converting to business metrics like cost-per-customer-interaction.
- Reporting agent costs in isolation without comparing to the manual processes they replace, reducing the risk of stakeholders evaluating automation ROI.
- Restricting cost data access to engineering teams, creating bottlenecks that delay optimization decisions.
- Presenting only quantitative dashboards without qualitative context, requiring business stakeholders to rely on engineering to interpret cost changes and recommend actions.

Benefits of establishing this best practice:

- Business-aligned metrics enable non-technical stakeholders to evaluate agent investments using familiar frameworks.
- ROI comparison against manual processes demonstrates automation value and justifies continued investment.
- Self-service cost dashboards reduce dependency on engineering for cost analysis, accelerating optimization decisions.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Business metrics work together with technical telemetry, but they require a translation layer to make sense to stakeholders. [Amazon Bedrock AgentCore Observability](#) produces the raw data (session count, token usage, execution duration) through Amazon CloudWatch integration, and [Amazon Bedrock AgentCore Identity](#) tags resources with business dimensions (like business-unit, product-line, and customer-segment). Enabling AWS Cost Explorer tag-based cost allocation

generates per-business-unit reports. The translation layer converts those technical units into cost-per-customer-interaction, cost-per-automated-decision, and cost-per-business-outcome. Those are the units that stakeholders can compare against other investments.

ROI demonstration needs a baseline cost model for the manual process the agent replaces: handling time, fully-loaded labor cost, error rate, and throughput limitations. The ROI calculation is the delta between that baseline and the agent's actual cost. Executives may not read CloudWatch dashboards, so build a BI layer with [Amazon Quick](#) fed by [AWS Cost and Usage Reports](#) and AgentCore Observability data. CloudWatch dashboards remain the operational tool for engineering, while Amazon Quick becomes the executive-facing tool for chargeback and ROI.

Narrative generation makes cost reports useful for non-technical audiences. Use a small [Amazon Bedrock](#) model invoked weekly by AWS Lambda to produce plain-language summaries of cost drivers with specific optimization recommendations and quantified savings estimates. Schedule the narrative generation with Amazon EventBridge Scheduler and distribute through Amazon SNS to business unit owners. A monthly review cadence helps you share cumulative savings and recommendations with stakeholders.

Implementation steps

1. **Propagate business dimension tags:** Configure [Amazon Bedrock AgentCore Identity](#) with business-unit and product-line tags propagated through all agent resources for AWS Cost Explorer reporting.
2. **Build a baseline cost model:** Capture pre-automation process costs (like handling time, fully-loaded labor cost, error rate, and throughput) and implement ROI calculation logic comparing agent costs against the baseline.
3. **Translate technical costs to business metrics:** Implement a cost translation layer that maps raw invocation costs to cost-per-decision and cost-per-task-completion, capturing how many business outcomes each dollar of agent spending delivers.
4. **Build executive-facing BI dashboards:** Use [Amazon Quick](#) fed by [AWS Cost and Usage Reports](#) and [Amazon Bedrock AgentCore Observability](#) data, displaying ROI trends and cost allocation by business unit.
5. **Automate narrative generation:** Invoke a small [Amazon Bedrock](#) model weekly from AWS Lambda to produce plain-language cost summaries that explain cost drivers, optimization actions taken, and specific recommendations with estimated savings.
6. **Keep operational dashboards in CloudWatch:** Use Amazon CloudWatch dashboards for operational cost monitoring by engineering teams, separate from executive-facing reporting.

7. **Establish a monthly review cadence:** Share cost narratives and optimization recommendations with business stakeholders each month, closing the loop between reporting and action.

Resources

Related best practices:

- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)
- [AGENTCOST05-BP03 Design tenant-aware cost allocation for AaaS pricing models](#)
- [AGENTCOST07-BP03 Create systematic optimization feedback loops for continuous improvement](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Economics for agentic AI on AWS](#)
- [Preparing the business for agentic AI at scale](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)
- [Amazon Quick User Guide](#)
- [AWS Cost and Usage Reports](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Observability tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Quick](#)
- [AWS Cost and Usage Reports](#)
- [Amazon CloudWatch](#)
- [AWS Cost Explorer](#)

Agent discovery and deployment cost optimization

Teams that optimize agent lifecycle infrastructure through lightweight discovery, efficient versioning, and warm initialization help keep operational overhead from growing faster than the agent fleet itself. Agent infrastructure costs can escalate through inefficient discovery mechanisms, version proliferation, and cold start penalties.

AGENTCOST06: How do you optimize agent discovery, registry, and deployment costs?

Capability intent

- Agent discovery runs on consumption-based infrastructure, so registry costs track actual query and write activity rather than a fixed overhead for the fleet.
- Metadata caching serves most capability lookups without re-reading the registry, keeping read costs proportional to the rate of registry change rather than the rate of invocation.
- Agent versioning stores shared dependencies once and retires unused versions automatically, so storage costs stay contained as the agent fleet evolves.
- Deployments use weighted traffic routing on shared infrastructure, so blue/green and canary rollouts don't pay twice for parallel environments.
- Frequently invoked agents reuse warm sessions with cached initialization artifacts, and infrequent agents scale to zero to avoid idle capacity charges.
- Cold start rates, cache hit rates, and per-version storage are instrumented and reviewed, giving owners the signal to address cost drivers before they scale with the fleet.

Maturity levels

These levels summarize what each stage of maturity looks like for agent discovery and deployment cost optimization as a whole.

Level	Name	What it looks like
1	Initial	Agent discovery, versioning, and initialization are handled one-time. Teams deploy heavyweight service mesh

Level	Name	What it looks like
		or custom registries without evaluating consumption-based alternatives. Container versions accumulate in Amazon ECR without lifecycle policies, and cold starts are absorbed on every invocation because session lifetimes, caching, and session affinity are not configured. Cost visibility is limited to account-level billing, so expensive patterns grow silently as the fleet expands.
2	Emerging	Managed discovery such as Amazon Bedrock AgentCore Gateway is adopted for tool-based agent collaboration. Custom registries, where needed, use consumption-based storage such as Amazon DynamoDB on-demand capacity . Version retention policies are defined but enforced manually. Warm session patterns are applied to a small number of high-frequency agents, and cold start rates are tracked for those agents.

Level	Name	What it looks like
3	Defined	Gateway or a consumption-based registry is the standard, and metadata caching with configurable TTLs is used across the fleet. Amazon ECR lifecycle policies delete unused images automatically, and container base layers are shared so ECR deduplicates them across agent versions. Weighted endpoint routing is used for blue/green and canary deployments on Amazon Bedrock AgentCore Runtime . Persistent filesystem caching amortizes initialization across sessions, and cold start and initialization costs are measured per agent type through Amazon Bedrock AgentCore Observability .

Level	Name	What it looks like
4	Proactive	Discovery, versioning, and initialization cost metrics are dashboarded in Amazon CloudWatch and reviewed on a routine cadence. Canary promotion is automated using cost-per-task-completion alongside error rate and latency, driven from AWS Lambda promotion logic. Session affinity routing is implemented in the orchestration layer, and Amazon Bedrock AgentCore Memory is configured for lazy loading so agents fetch only essential startup context. Cost anomalies on discovery , deployment, or initialization trigger alerts automatically.

Level	Name	What it looks like
5	Optimized	Discovery, deployment, and initialization patterns are continuously refined based on operational data. Agent version vending is fully self-service with cost-aware guardrails, and cold start rates sit below 10% across the fleet. Retention, promotion, and warm pool policies are self-healing and driven by observed usage. The organization publishes reusable agent and container patterns internally, and shares benchmarks on discovery, versioning, and initialization cost efficiency across teams.

Common issues to watch for

- Organizations deploy heavyweight service mesh infrastructure for simple capability lookups, paying fixed monthly costs for features the agent fleet doesn't use.
- Teams put the registry on the critical path for every invocation, repeatedly re-reading capability metadata instead of caching it, so read costs grow linearly with invocation volume.
- Agent version retention runs without lifecycle policies, so every configuration change accumulates images in the container registry and storage costs grow indefinitely.
- Blue/green and canary deployments run full parallel fleets instead of routing a small traffic percentage, doubling compute cost during every rollout.
- Frequently invoked agents pay cold start costs on every invocation because session lifetimes, persistent filesystem caching, and session affinity are not configured.
- Discovery, versioning, and initialization costs are not instrumented, so expensive patterns such as scan-heavy queries, unused versions, and repeated cold starts grow silently as the fleet expands.

Best practices

- [AGENTCOST06-BP01 Implement lightweight discovery and registry for cost-effective collaboration](#)
- [AGENTCOST06-BP02 Cost optimize versioning and deployment through efficient artifact management](#)
- [AGENTCOST06-BP03 Design cost-efficient initialization through warm pools and caching](#)

AGENTCOST06-BP01 Implement lightweight discovery and registry for cost-effective collaboration

Deploying service-mesh infrastructure for agent discovery carries a fixed cost that doesn't scale down when traffic does. You can keep registry cost proportional to fleet size through managed discovery through tool exposure, consumption-based registries, and aggressive metadata caching.

Desired outcome:

- You use consumption-based infrastructure for agent discovery, charging only for actual operations.
- You serve repeated capability lookups from a metadata cache instead of the database.
- You keep costs proportional to fleet size through efficient indexing and batched writes.
- You monitor per-query costs so they don't grow silently as the fleet expands.

Common anti-patterns:

- Deploying managed service mesh for simple capability lookups when a NoSQL database with consumption-based pricing would suffice.
- Making repeated registry lookups for the same agent metadata on every invocation without caching.
- Using full-table scans instead of targeted queries with proper indexes, consuming unnecessary read capacity.
- Operating agent discovery registries without monitoring per-query costs, which scale with fleet size and grow silently as the fleet expands.

Benefits of establishing this best practice:

- Consumption-based registry avoids fixed infrastructure overhead, charging only for actual read and write operations.
- Metadata caching serves repeated capability lookups without database queries, avoiding most read charges.
- Batched capability updates reduce write costs by accumulating changes into single operations.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Agent discovery through [Amazon Bedrock AgentCore Gateway](#) removes the need for custom registry infrastructure in many scenarios. When you expose specialized agents as tools through Gateway's MCP server capabilities, other agents discover and invoke them without a separate registry. AgentCore Gateway handles credential exchange, protocol translation, and composition of multiple tools into unified endpoints under consumption-based pricing. This approach fits when agents primarily interact through tool invocation patterns and Gateway's built-in discovery semantics match your collaboration requirements.

When those conditions don't hold, you can build a custom lightweight registry using [Amazon DynamoDB](#) with on-demand capacity mode and global secondary indexes for filtered capability queries. Index agent capabilities by category and task type so discovery queries read only relevant partitions rather than scanning the full registry. As agent fleets grow, inefficient discovery queries that scan entire capability sets multiply costs and degrade latency at the same time, so targeted queries through global secondary indexes become the operational baseline, not an optimization.

Metadata caching helps you cost optimize a registry designed for scalability. A capability lookup that hits the cache costs effectively nothing, while a lookup that falls through to DynamoDB incurs a read charge. Configure TTLs that reflect how often capability metadata actually changes (typically hours, not seconds), and use the persistent filesystem on [Amazon Bedrock AgentCore Runtime](#) to cache registry metadata across session stop and resume cycles.

Monitoring becomes increasingly important, as registry read costs scale with query volume, and per-query charges accumulate as the fleet grows. [Amazon Bedrock AgentCore Observability](#) and Amazon CloudWatch track read costs, cache hit rates, and discovery API call volumes, with alarms for anomalous patterns before they become a line item worth investigating.

Implementation steps

1. **Evaluate Gateway-based discovery first:** Determine whether [Amazon Bedrock AgentCore Gateway](#) semantic tool selection meets your discovery requirements. If so, expose agents through Gateway's MCP server capabilities and avoid the custom registry entirely.
2. **Use consumption-based storage with efficient indexing:** For custom registries, use [Amazon DynamoDB](#) with on-demand capacity mode. Design a global secondary index on capability-category so filtered queries by capability type or task category read only relevant partitions.
3. **Deploy a metadata cache:** Configure TTLs appropriate to how often capability metadata changes, and use the [Amazon Bedrock AgentCore Runtime](#) persistent filesystem to cache across session cycles.
4. **Monitor registry costs:** Track read costs and cache hit rates using [Amazon Bedrock AgentCore Observability](#) and Amazon CloudWatch, with alarms for anomalous patterns.

Resources

Related best practices:

- [AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations](#)
- [AGENTCOST06-BP02 Cost optimize versioning and deployment through efficient artifact management](#)
- [AGENTCOST06-BP03 Design cost-efficient initialization through warm pools and caching](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Observability](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Gateway](#)
- [AWS 2025 - AgentCore Registry: Discover, Govern, and Reuse AI Agents at Scale](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Gateway tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon DynamoDB](#)
- [Amazon CloudWatch](#)

AGENTCOST06-BP02 Cost optimize versioning and deployment through efficient artifact management

Agent runtimes that create an immutable version on every configuration change accumulate hundreds of versions during normal development, each holding references to container images that can't be managed easily. Layered base images, endpoint-based traffic routing, and automated cleanup policies keep deployment cost proportional to real usage rather than to the number of past configurations.

Desired outcome:

- You have container layer deduplication storing shared dependencies once across agent versions.
- You use endpoint-based traffic routing for blue/green and canary deployments without duplicate running infrastructure.
- You have automated lifecycle policies that delete unused versions, helping prevent indefinite storage accumulation.
- You monitor version inventory and catch unused versions before they become a material cost.

Common anti-patterns:

- Retaining all agent versions indefinitely without lifecycle policies, accumulating storage costs for versions that receive zero invocations.
- Creating separate container images without sharing common base layers, multiplying storage costs across agent versions.
- Running full parallel environments for blue/green deployments instead of routing only test traffic percentages.
- Allowing unused agent versions to accumulate without monitoring, reducing the risk of automated cleanup and steadily increasing storage overhead without visibility into version invocation patterns.

Benefits of establishing this best practice:

- Container layer deduplication stores shared dependencies once, reducing storage costs proportionally to version reuse. Verify deduplication by comparing total repository storage in ECR (reported in CloudWatch metrics under RepositorySize) against the sum of individual image manifest sizes. Effective deduplication shows repository storage significantly smaller than the sum of manifests.
- Endpoint-based routing enables deployment transitions without duplicate running infrastructure.
- Automated cleanup deletes unused versions, helping prevent indefinite storage cost growth.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

[Amazon Bedrock AgentCore Runtime](#) creates an immutable version on every configuration update. The version itself is lightweight metadata (container image reference, protocol settings, and network configuration), but each version holds a reference to container images that [Amazon ECR](#) can't be managed until all references are removed. Updating an environment variable, changing a protocol setting, or modifying network configuration all trigger a new version. Without a cleanup policy, active development produces hundreds of versions, each anchoring its referenced images in place.

The first mitigation is layer structure. Build agent containers with a common base layer containing shared dependencies (runtime, SDK, common tools) and agent-specific layers on top. ECR automatically deduplicates identical layers across images, so agents that share most dependencies share most storage. The second mitigation is traffic routing. The AgentCore Runtime endpoint system supports blue/green and canary deployments without parallel infrastructure: create a production endpoint pointing to the stable version and use weighted routing to send a small traffic percentage to new versions during validation. [Amazon Bedrock AgentCore Observability](#) provides the per-version metrics (error rates, latency percentiles, cost-per-task-completion) that drive the promotion decision.

Traditional canary promotion criteria look at error rates and latency, but agent versions can differ significantly on reasoning cost (a new prompt might produce correct answers that take 30% more tokens). Including cost-per-task-completion in the promotion criteria helps prevent a cost regression from slipping into production behind good quality metrics.

Define a maximum version retention (a reasonable starting point is the last five versions plus any version currently serving traffic through an endpoint) and configure [Amazon ECR lifecycle policies](#) to delete untagged images and images older than 90 days not referenced by active versions. Automated deletion of versions beyond the retention limit, gated on verification that no endpoints reference the version and it has had zero invocations during the retention window, keeps ECR from turning into a graveyard of development iterations.

Implementation steps

1. **Structure containers for layer sharing:** Build agent containers with common base layers and agent-specific layers so [Amazon ECR](#) layer deduplication is effective.
2. **Use endpoint-based traffic routing:** Deploy agents to [Amazon Bedrock AgentCore Runtime](#) and configure custom endpoints for production traffic with weighted routing for blue/green and canary deployments.
3. **Include cost in promotion criteria:** Monitor deployment quality using [Amazon Bedrock AgentCore Observability](#) metrics before updating production endpoints, including cost-per-task-completion alongside error rates and latency.
4. **Set version retention policies:** Define a retention policy such as the last five versions plus active traffic, and configure [Amazon ECR lifecycle policies](#) to delete unused images automatically.
5. **Monitor version inventory weekly:** Deploy a weekly version inventory function that queries AgentCore Runtime APIs for all agent versions, identifies versions with zero invocations through Amazon CloudWatch metrics, and stores the usage metadata for historical analysis before the ECR lifecycle policy deletes the images.

Resources

Related best practices:

- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)
- [AGENTCOST06-BP01 Implement lightweight discovery and registry for cost-effective collaboration](#)
- [AGENTCOST06-BP03 Design cost-efficient initialization through warm pools and caching](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)

- [Amazon ECR lifecycle policies](#)
- [Amazon Bedrock AgentCore Observability](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Runtime](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Runtime tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon ECR](#)
- [Amazon CloudWatch](#)

AGENTCOST06-BP03 Design cost-efficient initialization through warm pools and caching

Cold starts are a significant per-invocation cost in agent infrastructure, where model loading, tool registration, and memory hydration run on every fresh session. You can reduce those per-invocation costs through persistent filesystems, session affinity, and lazy context loading to reuse results while still scaling to zero for agents that are rarely called.

Desired outcome:

- You amortize initialization costs across many invocations by caching artifacts on a persistent filesystem.
- You have frequently invoked agents reusing warm sessions and infrequent agents scaling to zero without idle charges.
- You defer non-essential context retrieval to on-demand, keeping initialization under a fixed time budget.
- You track cold start rates and initialization costs per agent type.

Common anti-patterns:

- Performing expensive initialization on every invocation instead of caching artifacts across sessions.
- Allowing frequently invoked agents to repeatedly incur cold starts without session persistence or warm pool patterns.
- Loading all potentially relevant context at startup instead of lazy-loading on demand, increasing initialization latency with data that may never be used.
- Operating agents without cold start visibility, missing opportunities to apply warm pool patterns or optimize initialization for high-impact agents. Agent cold starts include model loading, tool registration, and memory hydration, not just container startup.

Benefits of establishing this best practice:

- Persistent filesystem caching amortizes initialization costs across many invocations, avoiding repeated overhead.
- Session lifecycle management maintains warm sessions for frequent agents while scaling to zero for infrequent ones without idle charges.
- Lazy context loading reduces initialization time by deferring non-essential retrieval until reasoning requires it.

Level of risk exposed if this best practice is not established: High

Implementation guidance

[Amazon Bedrock AgentCore Runtime](#) implements warm pool behavior through its session lifecycle. Sessions move through Active, Idle, and Terminated states. Active handles processing, Idle maintains session readiness after inactivity timeout without compute charges, and Terminated ends the session at expiration or maximum lifetime. Frequently invoked agents keep warm sessions through the idle window while infrequent agents scale to zero, so you don't pay idle capacity for agents that are not being called. Tune idle timeout based on invocation frequency to balance responsiveness with session overhead.

The persistent filesystem makes initialization caching worthwhile to implement. The filesystem survives session stop and resume cycles for up to 14 days, so expensive initialization artifacts (model state, tool configurations, and preloaded reference data) can be computed once on the

first invocation and reused across many later sessions. The 14-day retention shapes your caching strategy: plan artifact refresh and cache invalidation to fit inside the window, so cached data never ages out silently and never gets stale beyond the refresh point.

Session affinity optimizes costs by keeping the same `runtimeSessionId` across related invocations so loaded models and cached tool configurations are reused. Implement session tracking in the orchestration layer so user workflows map to consistent session identifiers, and monitor session reuse rates to confirm routing is avoiding unnecessary initialization overhead. If cold start rates rise above 10%, investigate affinity or idle timeout misconfiguration before optimizing the initialization logic itself.

Consider *lazy context loading*, where data is fetched or parsed only when it is required. [Amazon Bedrock AgentCore Memory](#) supports retrieving only minimal startup context (the user's current task and immediate session history), deferring long-term memory and knowledge base retrieval until the agent actually needs the data. This keeps initialization time under a 2-second target covering model loading and tool registration. Monitor cold start rates and initialization costs through [Amazon Bedrock AgentCore Observability](#), build Amazon CloudWatch dashboards for initialization duration and session reuse by agent type, and review idle timeout configuration monthly based on observed invocation patterns.

Implementation steps

1. **Cache initialization artifacts on persistent storage:** Configure [Amazon Bedrock AgentCore Runtime](#) with persistent filesystem (surviving up to 14 days across session cycles) and implement initialization logic that checks for cached artifacts before performing expensive operations.
2. **Apply session affinity:** Maintain consistent `runtimeSessionId` values across related invocations so warm session state (loaded models, cached tool configurations) is reused.
3. **Defer non-essential context:** Configure [Amazon Bedrock AgentCore Memory](#) for lazy loading of startup context, deferring additional retrieval to on-demand during reasoning, with an initialization time target under 2 seconds.
4. **Monitor cold start rates per agent type:** Instrument agents with [Amazon Bedrock AgentCore Observability](#) to track session creation latency, initialization duration (model loading and tool registration), and cold start rates, with alarms for rates exceeding 10%.

Resources

Related best practices:

- [AGENTCOST02-BP03 Use intelligent caching to reduce redundant model invocations](#)
- [AGENTCOST06-BP01 Implement lightweight discovery and registry for cost-effective collaboration](#)
- [AGENTCOST06-BP02 Cost optimize versioning and deployment through efficient artifact management](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Observability](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Runtime](#)
- [AWS 2025 - AgentCore Deep Dive: Memory](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Runtime advanced concepts](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Agent cost governance and continuous optimization

Organizations that implement layered cost controls (including budget enforcement, anomaly detection, and systematic optimization reviews) can deploy autonomous agents with spending bounded at multiple layers. Autonomous agents can generate unpredictable cost patterns that traditional budget controls don't address well.

AGENTCOST07: How do you establish agent cost governance and continuous optimization?

Capability intent

- Agent spending is bounded at multiple layers, including per-cycle, per-task, and per-day budgets, with automatic cutoffs that stop runaway reasoning and tool invocation before material cost is incurred.
- Agent-specific cost escalation patterns such as reasoning loop token spikes, tool invocation storms, and memory growth are detected within minutes of occurring, rather than days later through billing data.
- Cost governance preserves agent autonomy by using graduated throttling and approval workflows rather than binary shutdown, so that agents continue to complete tasks under pressure at reduced throughput.
- Cost optimization is a continuous organizational practice, with monthly reviews, A/B-tested changes, and quantitative cost-quality trade-off analysis feeding quarterly improvement targets.
- Cost signals are translated into business value, with cost-per-decision, cost-per-task-completion, and return on investment reported alongside spending so that investment decisions are grounded in outcomes rather than raw spend.

Maturity levels

These levels summarize what each stage of maturity looks like for agent cost governance and continuous optimization as a whole.

Level	Name	What it looks like
1	Initial	Agent cost governance is informal and reactive. Agents are deployed without per-agent budget limits or automatic cutoffs, and cost overruns are discovered through monthly bills rather than real-time signals. Anomaly detection relies on generic infrastructure monitoring that misses agent-specific failure modes, and

Level	Name	What it looks like
		there is no standing review cadence for cost optimization.
2	Emerging	Account-level budgets and basic alerts are in place through AWS Budgets , and a small set of agent-level budget limits exist for the highest-spending agents. Baseline telemetry is collected through Amazon Bedrock AgentCore Observability , and AWS Cost Anomaly Detection flags gross billing deviations. Cost optimization happens as ad-hoc projects triggered by overruns rather than on a regular cadence.

Level	Name	What it looks like
3	Defined	Hierarchical budget limits (per-cycle, per-task, per-day) and automatic iteration and token cutoffs are enforced deterministically through Amazon Bedrock AgentCore Policy and Amazon Bedrock Guardrails. Amazon CloudWatch Anomaly Detection operates on AgentCore Observability metrics with 2σ and 3σ thresholds, and custom detectors catch reasoning loop spikes and tool invocation storms. A monthly cost optimization review runs against a standard agenda, and investigation runbooks are in place for common anomaly types.

Level	Name	What it looks like
4	Proactive	Cost governance is proactive and integrated with delivery. Graduated throttling preserves agent function under pressure, and approval workflows gate cost-impacting capability changes. A/B testing through Amazon Bedrock agent alias routing validates optimizations before promotion, and Amazon Bedrock AgentCore Evaluations quantifies cost-quality trade-offs. Cost gates in the CI/CD pipeline block regressions, and quarterly improvement targets are tracked against cost-per-decision and cost-per-task-completion.

Level	Name	What it looks like
5	Optimized	Cost governance is codified, continuously validated, and self-adjusting. Budget policies, cutoffs, and graduated throttling are enforced as code, and anomaly thresholds are refreshed automatically from rolling baselines. Optimization feedback loops run continuously, with A/B tests, cost-quality efficiency ratios, and runbooks feeding back into agent design so that recurring failure modes are engineered out. Cost, quality, and business outcomes are reported together, and the organization shares cost-engineering patterns externally and contributes to industry practice.

Common issues to watch for

- Agents are deployed to production without per-agent budget limits or automatic iteration and token cutoffs, leaving runaway reasoning loops and tool invocation storms as a primary cost failure mode.
- Cost controls are treated as mutually exclusive with agent autonomy, so that teams either grant unbounded spending authority or restrict agents so heavily that task completion suffers.
- Anomaly detection is configured on generic infrastructure metrics rather than on agent-specific signals such as token consumption per session, tool invocation frequency, and memory growth, which allows agent-driven escalation to hide in aggregate noise.

- Anomalies are detected but not routed. Teams respond to every alert the same way, and ad-hoc diagnostic work delays resolution because there are no runbooks differentiating reasoning-loop spikes, tool storms, and memory growth.
- Cost optimization is run as an occasional initiative rather than a continuous practice, so inefficiencies accumulate between projects and optimizations are promoted fleet-wide without A/B testing or cost-quality trade-off analysis.

Best practices

- [AGENTCOST07-BP01 Implement automated cost controls with intelligent cutoffs](#)
- [AGENTCOST07-BP02 Establish proactive anomaly detection for agent cost patterns](#)
- [AGENTCOST07-BP03 Create systematic optimization feedback loops for continuous improvement](#)

AGENTCOST07-BP01 Implement automated cost controls with intelligent cutoffs

Autonomous agents that invoke tools and accumulate memory without caps are a primary cost risk of agentic systems. Hierarchical budget limits, automatic cutoffs on runaway sessions, and graduated throttling help keep your costs bounded without forcing the agent to stop working at the first sign of pressure.

Desired outcome:

- You enforce per-cycle, per-task, and per-day budget limits as pre-invocation checks, not alerts after the fact.
- You have automatic cutoffs that halt reasoning loops at iteration or cost thresholds.
- You have graduated throttling that slows invocations as budgets approach limits rather than forcing binary shutdown.
- You require approval for capability expansions that materially increase cost profiles.

Common anti-patterns:

- Deploying agents without budget limits, causing unexpected cost overruns during production operations.

- Allowing agents to enter unbounded reasoning loops that consume tokens each cycle without progress toward completion.
- Permitting unbounded tool invocations and memory growth: agents autonomously invoke tools and accumulate memory, and without caps, costs grow unbounded. This is a primary cost risk of autonomous agents.
- Treating cost controls and agent autonomy as mutually exclusive, either restricting agents excessively or granting unlimited spending authority.

Benefits of establishing this best practice:

- Hierarchical budget limits (like per-cycle, per-task, and per-day) create multiple defensive barriers against cost overruns.
- Automatic cutoffs halt reasoning loops at configured thresholds, addressing one of the most expensive failure modes in autonomous systems.
- Graduated throttling is designed to preserve agent functionality at reduced throughput rather than forcing abrupt shutdown.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Implement cost controls outside the agent's control loop for reliable enforcement. [Amazon Bedrock AgentCore Policy](#) applies Cedar policies at the [Amazon Bedrock AgentCore Gateway](#) boundary, helping prevent agents from bypassing budget limits through prompt manipulation. [Amazon Bedrock Guardrails](#) complements this by enforcing topic-based restrictions that help prevent tangential reasoning chains, which reduces token waste from off-topic exploration.

Hierarchical budgets give you multiple defensive barriers:

- Per-cycle limits catch individual runaway loops
- Per-task limits catch aggregate work inside a single user request
- Per-day limits catch sustained elevated usage

Automatic cutoffs track both iteration count and cumulative token cost per session, halting reasoning loops when thresholds are exceeded. Tool invocation caps per session matter as a separate control because each tool call incurs both the external API cost and the token cost of

processing returned data. Uncapped tool use can drain the token budget from the other direction. Memory growth guardrails cap context window growth rate because every token in context is paid on every subsequent invocation, turning unbounded accumulation into a compounding cost driver.

Throttling is another useful automated control. Amazon API Gateway usage plans or custom Lambda-based rate limiting reduce maximum throughput as daily budgets approach limits, slowing token consumption without forcing hard cutoffs. Throttling and cutoffs operate at different scales. Throttling handles sustained high usage, while cutoffs handle individual runaway sessions. A well-designed control stack uses both, so normal high traffic is slowed rather than stopped, and pathological sessions are stopped rather than slowed.

Monitor these controls by using [Amazon Bedrock AgentCore Observability](#), which feeds Amazon CloudWatch dashboards for budget utilization, cutoff activations, and throttling events. For cost-impacting configuration changes (adding expensive tools, upgrading models, or expanding autonomous capabilities), integrate cost review gates into the CI/CD pipeline so significant cost impacts receive review before deployment.

Implementation steps

1. **Enforce budget limits through Cedar policies:** Configure [Amazon Bedrock AgentCore Policy](#) with Cedar policies enforcing per-cycle, per-task, and per-day budget limits at the Gateway boundary, including tool invocation caps per session and memory growth guardrails that trigger summarization when context approaches model limits.
2. **Deploy automatic cutoffs and topic guardrails:** Track iteration counts and cumulative costs per session with automatic cutoffs, and use [Amazon Bedrock Guardrails](#) for topic-based restrictions on tangential reasoning.
3. **Add graduated throttling:** Implement progressive throttling that slows invocations as budgets approach limits, keeping agent operations at reduced throughput rather than forcing binary shutdown.
4. **Visualize and alarm on governance metrics:** Create Amazon CloudWatch dashboards displaying budget utilization, cutoff activations, and throttling events using [Amazon Bedrock AgentCore Observability](#) metrics.

Resources

Related best practices:

- [AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops](#)

- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)
- [AGENTCOST07-BP02 Establish proactive anomaly detection for agent cost patterns](#)
- [AGENTCOST07-BP03 Create systematic optimization feedback loops for continuous improvement](#)

Related documents:

- [Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore Policy](#)
- [AWS Budgets](#)
- [Amazon Bedrock capacity, limits, and cost optimization](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)

Related videos:

- [AWS re:Invent 2024 - Balance cost, performance & reliability for AI at enterprise scale \(AIM3304\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Policy tutorials](#)

Related workshops:

- [Diving Deep into Bedrock AgentCore - Policy](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [AWS Budgets](#)

AGENTCOST07-BP02 Establish proactive anomaly detection for agent cost patterns

Generic billing alerts help you find cost escalation more quickly after it starts. Agent-specific anomaly detection catches reasoning loop token spikes, tool invocation storms, and memory growth quickly, and routing those alerts correctly means that you can alert the team that owns the agent instead of the operations team.

Desired outcome:

- You establish ML-based anomaly detection with statistical baselines and deviation thresholds.
- You have custom detectors for agent-specific failure modes beyond generic infrastructure monitoring.
- You pair every anomaly type with an investigation runbook to accelerate resolution.
- You correlate anomalies to route agent-driven issues to development teams and infrastructure issues to operations teams.

Common anti-patterns:

- Deploying anomaly detection without sufficient baseline data, generating excessive false positives that undermine team confidence.
- Relying solely on generic infrastructure monitoring that misses agent-specific failure modes driving the highest costs.
- Detecting anomalies without investigation runbooks, leaving costs escalating while teams figure out diagnostic procedures as issues occur.
- Treating all cost spikes as equivalent when agent spikes have different root causes (reasoning loops, tool storms, memory growth) that require different remediation.
- Collecting anomaly insights without feeding them back into agent design changes (tighter iteration limits, better prompts, tool caching) that help prevent recurrence.

Benefits of establishing this best practice:

- Proactive detection identifies cost escalation from agent-specific failure modes within minutes rather than days.

- Investigation runbooks reduce mean time to resolution by replacing ad-hoc analysis with guided diagnostic execution.
- Correlation analysis routes alerts to the right team (agent development or operations), helping prevent triage delays.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Anomaly detection needs real baselines before it is useful. Collect 2 to 4 weeks of baseline operational data before setting thresholds, because detectors configured on insufficient history produce false positives that erode confidence in the whole system. [Amazon CloudWatch Anomaly Detection](#) automatically learns statistical baselines for agent cost metrics and generates dynamic anomaly bands that adapt to seasonality and trends. Apply it to [Amazon Bedrock AgentCore Observability](#) metrics including token consumption per session, tool invocation frequency, memory growth, and cost-per-task-completion. Use 2σ for warning and 3σ for critical alerts.

Generic infrastructure monitoring doesn't catch the failure modes that cost agents the most money. Reasoning loop token spikes, tool invocation storms, and memory growth are agent-specific patterns, and they need agent-specific detectors. Reasonable initial thresholds include:

- Reasoning loop token spikes at 5x session average
- Tool invocation storms at 3x baseline rate
- Memory storage growth at 2x per hour
- Multi-agent workflow cost escalation at 2x historical average

These catch pathological behavior early enough to help prevent material cost impact.

Correlation analysis helps make routing a sensible choice. An agent-driven anomaly correlates with specific agent IDs in cost allocation tags and shows up in AgentCore Observability token consumption or invocation patterns. An infrastructure anomaly happens independently of agent behavior and shows up in generic service metrics. Routing agent-driven anomalies to development teams (with context about which reasoning pattern triggered the spike) and infrastructure anomalies to operations teams (with context about constrained resources) keeps alerts in front of the people who can act on them. AgentCore Observability span analysis drills further. Is the spike in planning tokens, tool calls, or memory growth? That determines whether the fix is a prompt change, a tool cache, or a tighter memory policy.

[AWS Cost Anomaly Detection](#) provides a billing-level backstop. Configured per agent cost allocation tag, it catches gradual escalations that are not visible in operational metrics. Investigation runbooks for each anomaly type (diagnostic queries, likely root causes, and immediate mitigation actions) live in AWS Systems Manager OpsCenter, with CloudWatch Logs Insights queries for traces, AWS X-Ray for distributed workflows, and AgentCore Observability span analysis for token patterns.

Implementation steps

1. **Baseline, then detect:** Collect 2 to 4 weeks of baseline data using [Amazon Bedrock AgentCore Observability](#), then configure [Amazon CloudWatch Anomaly Detection](#) on key cost indicators with 2σ warning and 3σ critical thresholds.
2. **Implement agent-specific detectors:** Deploy Lambda-based detectors for reasoning loop token spikes (5x session average), tool invocation storms (3x baseline rate), memory growth anomalies (2x per hour), and workflow cost escalation (2x historical average), publishing structured anomaly events to Amazon EventBridge.
3. **Add billing-level anomaly coverage:** Configure [AWS Cost Anomaly Detection](#) monitors per agent cost allocation tag as a backstop for gradual escalations.
4. **Create investigation runbooks:** Store runbooks for each anomaly type in AWS Systems Manager OpsCenter, with diagnostic queries (Amazon CloudWatch Logs Insights for traces, AWS X-Ray for distributed workflows, AgentCore Observability span analysis for token patterns) and mitigation actions.
5. **Route anomalies through correlation analysis:** Classify anomalies using cost allocation tags and AgentCore Observability dimensions, routing agent-driven anomalies to development teams and infrastructure anomalies to operations teams.

Resources

Related best practices:

- [AGENTCOST01-BP01 Use the reflection pattern to design efficient agent reasoning loops](#)
- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)
- [AGENTCOST07-BP01 Implement automated cost controls with intelligent cutoffs](#)
- [AGENTCOST07-BP03 Create systematic optimization feedback loops for continuous improvement](#)

Related documents:

- [Amazon CloudWatch Anomaly Detection](#)
- [AWS Cost Anomaly Detection](#)
- [Amazon Bedrock AgentCore Observability](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Observability tutorials](#)

Related services:

- [Amazon CloudWatch](#)
- [AWS Cost Management](#)
- [Amazon Bedrock AgentCore](#)

AGENTCOST07-BP03 Create systematic optimization feedback loops for continuous improvement

One-time optimization projects can miss compounding gains, as cost-performance characteristics of agent systems shift as prompts change, tools evolve, and traffic patterns drift. A monthly review cadence, A/B-tested changes, and cost gates in the deployment pipeline turn optimization into a continual practice that can keep pace with the system.

Desired outcome:

- You hold monthly cost optimization reviews following a structured agenda.
- You A/B test cost-impacting changes through controlled traffic routing before fleet-wide promotion.
- You calculate cost-quality efficiency ratios to prioritize optimizations by business value.
- You have cost gates in deployment pipelines helping prevent accidental regressions.

Common anti-patterns:

- Cutting costs without measuring quality impact, degrading agent performance and undermining business value.

- Treating cost optimization as an occasional initiative without regular cadence, allowing inefficiencies to accumulate.
- Promoting optimizations across the entire fleet without A/B testing, exposing all users to potential quality degradation.
- Tracking costs without correlating them to business outcomes or setting quantitative improvement targets, reducing the risk of data-driven prioritization.

Benefits of establishing this best practice:

- Systematic review cycles continually identify high-impact optimization opportunities with accountability for progress.
- A/B testing validates optimization hypotheses before deployment, helping prevent costly mistakes from untested changes.
- Cost gates in deployment pipelines block changes that increase costs without corresponding capability improvements.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Monthly reviews drive continual optimization. Structure it with four sections:

1. Cost efficiency trends (cost-per-decision, cost-per-task-completion against targets)
2. Top optimization opportunities ranked by impact and effort
3. Experiment results from the previous month
4. Next-month planning

[Amazon Bedrock AgentCore Observability](#) provides granular tracing that can reveal opportunities invisible in aggregate metrics. AgentCore Runtime tracing decomposes spend by reasoning pattern, tool invocation, and memory operation, so you can see where to act rather than just that action is needed. Set quarterly improvement targets and track progress in Amazon CloudWatch dashboards and AWS Cost Explorer.

A/B testing is critical for continual optimizations. Use [Amazon Bedrock agent alias routing](#) to split traffic between control and treatment configurations. For each experiment, define a hypothesis, success metrics covering both cost and quality, and a minimum observation period for statistical

significance. For agent workloads where task completion quality varies more than in traditional request-response patterns, plan on at least a 1-week observation, a 10% traffic split, and at least 1,000 task completions before calling a result. [Amazon Bedrock AgentCore Evaluations](#) runs standardized quality assessments before and after each optimization so the cost reduction isn't accompanied by a quality regression.

Prioritization needs a decision framework. Calculate a cost-quality efficiency ratio as cost reduction percentage divided by quality degradation percentage. Ratios above 10 indicate strong opportunities where most of the quality is preserved per dollar saved, and ratios below 2 signal that quality degradation is too large relative to cost savings and should be deprioritized. This framework lets reviewers rank candidate optimizations consistently against each other.

[Amazon Bedrock AgentCore Memory](#) cost allocation features identify memory-driven costs that should inform retention policy changes. When expanding agent tool capabilities through [Amazon Bedrock AgentCore Gateway](#), measure tool invocation frequency and reasoning cost before promotion rather than after. [Amazon Bedrock AgentCore Policy](#) provides deterministic constraints if cost analysis shows agents over-invoking expensive tools. Cost gates in the CI/CD pipeline compare estimated cost-per-task against the current version and block deployment when the increase exceeds threshold without a corresponding capability improvement.

Implementation steps

1. **Run a monthly optimization review:** Use Amazon CloudWatch dashboards and AWS Cost Explorer for trend visualization, with a structured agenda covering efficiency trends, opportunities, experiment results, and next-month plans.
2. **A/B test cost-impacting changes:** Use [Amazon Bedrock agent alias routing](#) with defined hypotheses and success criteria covering both cost and quality. Configure 10% traffic splits, 1-week minimum observation periods, and 1,000 task completion minimums for statistical significance.
3. **Prioritize with cost-quality ratios:** Use [Amazon Bedrock AgentCore Evaluations](#) to measure cost-quality trade-offs and calculate efficiency ratios. Target ratios above 10 for strong opportunities and deprioritize ratios below 2.
4. **Analyze phase and memory costs:** Use [Amazon Bedrock AgentCore Observability](#) Runtime tracing to identify per-phase reasoning cost patterns, and [Amazon Bedrock AgentCore Memory](#) cost allocation to tune retention policies.

5. **Assess tool expansion cost before promotion:** When adding capabilities through [Amazon Bedrock AgentCore Gateway](#), measure tool invocation frequency and reasoning cost before promotion, and use [Amazon Bedrock AgentCore Policy](#) to constrain expensive tool usage.
6. **Add cost gates to CI/CD:** Block deployments that exceed cost increase thresholds without corresponding capability improvements.
7. **Set quarterly improvement targets:** Track progress against targets in the monthly review so optimization is measurable, not aspirational.

Resources

Related best practices:

- [AGENTCOST02-BP02 Cost optimize token consumption through efficient prompt engineering](#)
- [AGENTCOST05-BP01 Establish agent-level reasoning cost tracking and attribution](#)
- [AGENTCOST05-BP04 Create chargeback and ROI reporting](#)
- [AGENTCOST07-BP01 Implement automated cost controls with intelligent cutoffs](#)
- [AGENTCOST07-BP02 Establish proactive anomaly detection for agent cost patterns](#)

Related documents:

- [Economics for agentic AI on AWS](#)
- [Effective cost optimization strategies for Amazon Bedrock](#)
- [Operationalizing agentic AI on AWS](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Policy](#)
- [Guidance for Cost Analysis and Optimization with Amazon Bedrock Agents](#)

Related videos:

- [AWS re:Invent 2024 - Sustainable and cost-efficient generative AI with agentic workflows \(AIM333\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Observability tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)
- [AWS Cost Explorer](#)

Sustainability

Organizations that build sustainable agent practices from the start scale their automation investments confidently, preserving institutional expertise, maximizing resource efficiency, and delivering compounding value as agent portfolios grow. Agentic AI systems introduce new sustainability challenges as agents consume compute, memory, and cognitive resources across extended workflows. This pillar addresses how organizations can build agent systems that maximize resource efficiency, minimize waste through reusable architectures, and maintain sustainable adoption patterns that preserve organizational knowledge and meet long-term business needs.

Capabilities

- [Resource reusability](#)
- [Resource right-sizing](#)
- [Agent governance](#)

Design principles

In addition to the lens-level design principles, the sustainability best practices in this lens are represented by at least one of the following principles:

- **Right-size each agent to its capability:** Resource boundaries (timeout, memory, token budget) match the actual work of a single capability rather than the worst-case input across many. For long-running or exploratory agents, declared success criteria, cost ceilings, and termination conditions keep extended execution a bounded investment instead of open-ended drift.
- **Amortize platform investment across the fleet:** Reusable workflow patterns, shared infrastructure services (connection pools, vector stores, retrieval pipelines, queues), and fleet-wide caching of model calls, tool results, and memory lookups turn what every team would otherwise build, run, and pay for separately into one piece of infrastructure every team uses. The first invocation does the work; subsequent ones reuse it.
- **Scale cognitive and infrastructure intensity to observed demand:** Match model size, retrieval depth, and memory scope to task complexity at the reasoning layer, and size hosting, network, and storage to the bursty, variable shape of real agent traffic instead of a theoretical maximum at the platform layer. Default-large is the most expensive path against any resource — energy, carbon, water, and silicon included.

- **Measure environmental footprint as an engineering signal:** Track energy, carbon, and water impact alongside cost and latency, baseline it per agent and per workflow, and route optimization investment to where the data points. Sustainability claims that aren't measured are aspirations, and optimizations that aren't measured are guesses.
- **Sustain the human capability that automation amplifies:** Maintain the organizational skills, processes, and tacit knowledge behind every automated workflow, and target agent development at work the organization has already mastered rather than processes it hopes to learn through automation. Agents that codify proven work pay back; agents that try to substitute for missing expertise consume resources discovering what could have been documented.
- **Curate the agent portfolio through living specifications and active retirement:** Keep specifications current as the system of record for each agent's purpose, scope, integration points, and decision logic so institutional knowledge survives staff and tooling changes, and decommission agents that no longer earn their infrastructure, security surface, and operational overhead before the portfolio becomes its own sustainability problem.

Resource reusability

Organizations that invest in reusable agent infrastructure accelerate every subsequent project while reducing cumulative resource consumption. As agents are used more, the opportunity to optimize, share, and recycle agent resources increases. Shared resources pose a less significant burden on the overall architecture, creating a more sustainable workflow.

AGENTSUS01: How do you build sustainable and repeatable frameworks for managing compute, memory, and other shareable agent resources?

Capability intent

- Each agent encapsulates a single, well-defined capability with explicit resource boundaries, and those boundaries cascade through delegation to child agents so no subtree consumes resources without a defined cap.
- Recurring workflow patterns (retrieval, validation, transformation, and decision-making) exist as parameterized, discoverable components that teams compose rather than rebuild for every new use case.

- Common infrastructure such as connection pools, caches, authentication, and foundation model access is consolidated into shared services, so capacity grows with aggregate demand rather than with the number of agent instances.
- Cognitive pathways are right-sized to task complexity, with model selection, retrieval depth, and memory scope each matched to actual needs rather than defaulting to the largest or broadest option.
- Long-running and frontier workloads run against explicit specifications covering success criteria, resource budgets, and termination conditions, with checkpoints that make compute investment reviewable against business value.

Maturity levels

These levels summarize what each stage of maturity looks like for resource reusability as a whole.

Level	Name	What it looks like
1	Initial	Agents are provisioned for worst-case scenarios without explicit resource contracts, and delegated child agents inherit no constraints. Workflows are rebuilt from scratch per use case, with no shared pattern library or inventory of reusable components. Every agent establishes its own connections to external services, duplicating infrastructure across instances. Foundation model calls default to the largest available model, and long-running agents run without specifications or checkpoints.

Level	Name	What it looks like
2	Emerging	Teams have started to decompose workflows into specialized agents with basic timeout and memory limits configured per agent. Some recurring workflow patterns have been extracted into templates, though discovery is informal. A handful of shared services (usually a central cache or a common authentication path) are in production. Amazon Bedrock AgentCore Observability is enabled for most production agents, and manual reviews of resource consumption occur on a regular cadence.

Level	Name	What it looks like
3	Defined	Specialized agents are the default pattern, deployed on Amazon Bedrock AgentCore Runtime with explicit resource boundaries, and reusable workflow patterns are published through Amazon Bedrock AgentCore Gateway as discoverable MCP tools. Shared caches such as Amazon ElastiCache are standard, and authentication runs through Amazon Bedrock AgentCore Identity. Tiered model routing and scoped retrieval against Amazon Bedrock Knowledge Bases are in place for most workflows, and long-running agents run with documented specifications.

Level	Name	What it looks like
4	Proactive	Resource budgets are enforced at the orchestration layer through AWS Step Functions and AgentCore Policies rather than relying on agent self-restraint. Pattern libraries are maintained, governed, and instrumented so a single optimization propagates to every consuming workflow. Amazon Bedrock cross-region inference distributes model traffic, and Amazon Bedrock Data Automation handles document extraction in place of large vision models. Frontier agents run with cascading budgets, checkpoint-based evaluation, and state persistence through Amazon Bedrock AgentCore Memory.

Level	Name	What it looks like
5	Optimized	Resource boundaries, routing thresholds, retention policies, and specification templates are continuously recalibrated from Amazon Bedrock AgentCore Observability data rather than through manual review cycles. Shared services and pattern libraries are the default composition point for new workflows, and new development that doesn't reuse existing components is the exception. Resource utilization efficiency (successful completions per unit of compute) is a first-class metric alongside cost and latency, and the organization contributes reusable patterns and sustainability measurements back to its communities of practice.

Common issues to watch for

- Teams provision agents for worst-case scenarios and let parent agents delegate work to child agents without cascading resource budgets, so runaway consumption in one subtree is discovered only after the monthly bill.
- Recurring workflow logic is rebuilt for each new use case, accumulating duplicated code and testing debt while missing the renewable-architecture benefit of optimizing a shared pattern once and propagating the improvement everywhere.

- Each agent establishes its own connections, caches, and authentication flows rather than consuming shared services, so infrastructure costs scale linearly with agent count instead of with aggregate demand.
- Every request is routed to the largest foundation model, retrieval is unbounded against knowledge bases, and memory grows without pruning, so cognitive overhead is disproportionate to the value delivered by each interaction.
- Long-running and frontier agents run without specifications, resource budgets, or checkpoints, so they explore tangential paths and consume extended compute without an accountability mechanism that ties outcomes to business value.

Best practices

- [AGENTSUS01-BP01 Design specialized agents with explicit resource boundaries](#)
- [AGENTSUS01-BP02 Implement reusable workflow patterns](#)
- [AGENTSUS01-BP03 Optimize resource utilization through shared services](#)
- [AGENTSUS01-BP04 Scale cognitive processing pathways appropriately](#)
- [AGENTSUS01-BP05 Adopt specification-driven tasks for frontier agents and long-running workflows](#)

AGENTSUS01-BP01 Design specialized agents with explicit resource boundaries

Monolithic agents that over-provision for worst-case inputs waste compute without reliable audit trails to track consumption. To make consumption traceable at every layer, use specialized agents with single atomic capabilities and explicit resource boundaries. Give each agent a timeout, a memory ceiling, and a token budget. Each unit of resource spend then maps back to the task that caused it.

Desired outcome:

- You have decomposed workflows into specialized agents, each responsible for one atomic capability with declared resource limits.
- Parent agents cascade resource budgets to child agents through the orchestration layer, so delegation has predictable compute and token costs.
- You track resource consumption for each agent (duration, tokens, and error rates) across delegation chains so over-provisioning is visible.

- Reusable specialized agents are exposed through a shared tool layer, so one well-bounded agent serves many parent workflows.

Common anti-patterns:

- Provisioning compute and memory for worst-case inputs regardless of actual task requirements, producing low utilization and unnecessary cost.
- Delegating from parent agents to child agents without passing timeout, retry, or token budgets, so downstream work has no enforceable cost ceiling.
- Deploying monolithic agents that bundle multiple capabilities in one process, which prevents independent scaling and makes per-capability cost attribution infeasible.
- Duplicating implementations of a capability (validation, extraction, or transformation) across workflows because no shared agent exists with known resource bounds.

Benefits of establishing this best practice:

- Resource consumption stays proportional to the task, because each agent runs within bounds appropriate to its single capability.
- Cost attribution is visible at the agent level. Over-provisioned or underperforming agents are straightforward to identify and right-size.
- Specialized agents amortize their development cost across many parent workflows when exposed as reusable tools.

Level of risk exposed if this best practice is not established: High

Implementation guidance

The right unit of resource accountability is the capability, not the deployment. When a single process handles validation, enrichment, extraction, and decision-making, the only safe way to size it is to assume every call does all four. Splitting those capabilities into separate agents lets each one carry the timeout, memory ceiling, and token budget that fit its actual work. Right-sizing becomes a question about each capability rather than a compromise across the whole workflow.

Budgets stop being useful the moment a delegation crosses a boundary without carrying them along. The orchestration layer has to propagate remaining time, remaining tokens, and retry budget into every child invocation. In [AWS Step Functions](#), that means setting `TimeoutSeconds` and

retry counts on each nested state. In a Strands-based orchestrator, it means passing the remaining budget as part of the child invocation parameters. Without that cascade, total workflow cost is unbounded regardless of what the top-level agent promises.

When a single well-bounded data validation agent serves dozens of parent workflows through [Amazon Bedrock AgentCore Gateway](#) MCP server capabilities, its development and optimization cost is amortized across every caller. [Amazon Bedrock AgentCore Runtime](#) provides the session-isolated execution environment that makes each invocation carry its own resource context. [Amazon Bedrock AgentCore Policy](#) acts at the traffic boundary to reject invocations that exceed declared limits before they consume capacity.

[Amazon Bedrock AgentCore Observability](#) captures duration, token counts, and error rates for each invocation across delegation chains, so the utilization picture for each agent is the same at every level of the hierarchy. Review consumption by agent monthly to find agents that consistently run well below their declared limits. Those are the first candidates for right-sizing.

Implementation steps

- 1. Decompose workflows into single-capability agents:** Identify atomic functions in each workflow and deploy each as its own agent on [Amazon Bedrock AgentCore Runtime](#) with explicit timeout, memory, and token limits. Common atomic functions include:
 - Validation
 - Extraction
 - Transformation
 - Decision
- 2. Cascade budgets across delegation boundaries:** Configure the orchestration layer ([AWS Step Functions](#) or a Strands-based orchestrator) to pass the following into every child invocation so downstream work inherits the parent's cost ceiling:
 - Remaining time
 - Retry count
 - Token budget
- 3. Expose specialized agents as reusable tools:** Publish agents through [Amazon Bedrock AgentCore Gateway](#) MCP server capabilities so parent workflows invoke them without each embedding its own copy.

4. **Enforce limits at the traffic layer:** Apply [Amazon Bedrock AgentCore Policy](#) Cedar rules at the Gateway boundary to reject invocations that exceed declared resource limits before they consume capacity.
5. **Instrument consumption and review monthly:** Turn on [Amazon Bedrock AgentCore Observability](#) to capture the following for each agent:
 - Invocation duration
 - Token counts
 - Error rates

Review utilization in Amazon CloudWatch each month to right-size boundaries against actual usage.

Resources

Related best practices:

- [AGENTSUS01-BP02 Implement reusable workflow patterns](#)
- [AGENTSUS01-BP03 Optimize resource utilization through shared services](#)
- [SUS03-BP01 Optimize software and architecture for asynchronous and scheduled jobs](#)
- [COST09-BP03 Supply resources dynamically](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon Bedrock AgentCore Policy](#)
- [Amazon Bedrock AgentCore Observability](#)
- [AWS Step Functions - Nested workflows](#)
- [Introducing Amazon Bedrock AgentCore Gateway: Transforming enterprise AI agent tool development](#)
- [Strands Agents](#)

Related examples:

- [Build multi-agent systems with LangGraph and Amazon Bedrock](#)

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Runtime tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Step Functions](#)
- [Amazon CloudWatch](#)

AGENTSUS01-BP02 Implement reusable workflow patterns

When every team rebuilds data retrieval, validation, and transformation workflows from scratch, each rebuild costs development time and pays for its own separate optimization cycle. A library of parameterized patterns shifts that cost into a one-time investment and makes subsequent projects compose from tested building blocks instead of starting over.

Desired outcome:

- You have a catalog of parameterized workflow patterns for recurring agent tasks (retrieval, validation, transformation, and decision-making) that teams can discover and instantiate.
- Teams compose new agent systems from existing patterns before writing new implementations.
- Each pattern has a documented interface, version history, and declared resource profile, and a single pattern serves many callers through a shared tool layer.
- You monitor pattern invocation frequency and failure rates so optimizations to a pattern propagate to every caller.

Common anti-patterns:

- Building single-use workflows for each new project without considering reuse, duplicating effort across teams and accumulating technical debt as similar capabilities are reimplemented with slight variations.
- Hardcoding workflow logic and decision points into individual implementations instead of parameterizing them, making reuse of proven patterns impractical.
- Developing reusable components without a central catalog or documentation, so teams rebuild workflows that already exist elsewhere in the organization.

- Treating workflows as disposable code rather than maintained assets, skipping the testing, documentation, and versioning that keeps patterns usable past their first deployment.

Benefits of establishing this best practice:

- Each new agent project starts from proven, tested building blocks instead of rebuilding from scratch, reducing the redundant development and test cycles that dominate early adoption.
- Behavior stays consistent across teams because every caller of a pattern gets the same validated implementation.
- A single optimization to a shared pattern improves every caller at once, amortizing future tuning work across the whole fleet.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

A pattern only becomes reusable when its parameters, inputs, outputs, and failure modes are explicit. The recurring workflows in agent systems, a retrieval chain with source selection, a validation pipeline with schema checks, and a transformation stage with format conversion, look the same across projects because the shape of the work is the same. The part that varies is narrow. It's which source, which schema, and which format. Making those parameters part of the interface, rather than hardcoding them into each implementation, allows one implementation to serve many callers.

Discovery has to work at two different times. At design time, a team evaluating whether to build something new needs to find out what already exists. That is a documentation repository problem. At runtime, an orchestrating agent needs to resolve a capability to a concrete endpoint. That is a registry problem. [Amazon Bedrock AgentCore Gateway](#) exposes patterns as MCP tools with well-defined interfaces, so runtime discovery is built in. The design-time catalog (a wiki page, a service catalog, or a README) has to be maintained alongside the registry so teams can browse capabilities, limitations, and resource requirements before writing code.

Access controls and versioning keep shared patterns from being forked silently by every team. Apply [Amazon Bedrock AgentCore Identity](#) and Gateway policies so pattern usage has a known footprint. Keep pattern versions explicit so consumers can adopt a new version deliberately. Deploy the underlying implementations on [Amazon Bedrock AgentCore Runtime](#) so each pattern has the same operational baseline.

[Amazon Bedrock AgentCore Observability](#) exposes invocation frequency, execution duration, and failure rates per pattern in Amazon CloudWatch, so owners can tell which patterns are heavily used (and therefore worth investing in) and which are sitting unused (and are probably the wrong abstraction or undiscoverable).

Implementation steps

- 1. Identify and extract recurring patterns:** Audit existing agent workflows for repeated sequences and pull each one out into a parameterized template with a documented interface. Common sequences include:
 - Retrieval
 - Validation
 - Transformation
 - Decision
- 2. Deploy patterns as reusable components:** Implement each pattern on [Amazon Bedrock AgentCore Runtime](#) and publish it through [Amazon Bedrock AgentCore Gateway](#) as an MCP tool so orchestrating agents can discover and invoke it at runtime.
- 3. Maintain a design-time catalog:** Keep a documentation repository alongside the Gateway registry where teams can browse the following before writing new code:
 - Pattern purpose
 - Parameters
 - Resource profile
 - Usage examples
- 4. Govern pattern access and versions:** Apply [Amazon Bedrock AgentCore Identity](#) and Gateway policies to control who can invoke each pattern, and publish new versions alongside old ones so consumers migrate deliberately.
- 5. Monitor usage and feed improvements back:** Track the following for each pattern through [Amazon Bedrock AgentCore Observability](#):
 - Invocation frequency
 - Duration
 - Failure rates

Use the telemetry to find heavily used patterns worth investing in and stagnant patterns worth retiring.

Resources

Related best practices:

- [AGENTSUS01-BP01 Design specialized agents with explicit resource boundaries](#)
- [AGENTSUS01-BP03 Optimize resource utilization through shared services](#)
- [SUS03-BP01 Optimize software and architecture for asynchronous and scheduled jobs](#)
- [OPS11-BP03 Iterate to improve](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway](#)
- [Introducing Amazon Bedrock AgentCore Gateway: Transforming enterprise AI agent tool development](#)
- [AWS Step Functions - State machine templates](#)
- [Strands Agents](#)
- [Guidance for Multi-Agent Orchestration on AWS](#)

Related videos:

- [AWS 2025 - Building AI Agents with Serverless, Strands, and MCP \(NTA405\)](#)

Related examples:

- [GitHub: aws-samples/aws-stepfunctions-examples](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Step Functions](#)

AGENTSUS01-BP03 Optimize resource utilization through shared services

Every agent that provisions its own connection pool, cache, or processing queue pays the cost of infrastructure nobody else in the fleet benefits from. Shared services turn those duplications into one piece of infrastructure that every agent uses, so infrastructure scales with organizational demand rather than agent count.

Desired outcome:

- You have common infrastructure, connection pools, caches, and processing queues, consolidated into shared service layers that every agent invokes rather than duplicates.
- Agents consume shared services through a tool abstraction, so implementations can change without coupling to the callers.
- Shared caching and pooling reduce the total number of redundant calls to external systems.
- Utilization of shared services is monitored so capacity scales with actual demand rather than theoretical peaks.

Common anti-patterns:

- Deploying a separate cache, connection pool, and queue per agent, so infrastructure cost scales linearly with agent count.
- Letting each agent open its own connections to external services and fetch the same reference data repeatedly, producing redundant network traffic and wasted compute.
- Treating each agent workflow as an isolated system, missing opportunities to consolidate common functions like authentication, logging, or queuing into a shared layer.
- Maintaining static allocations regardless of actual demand, so shared infrastructure carries peak capacity even during low-utilization periods.

Benefits of establishing this best practice:

- A single optimization to shared infrastructure improves every agent that uses it, amortizing operational work across the fleet.
- Infrastructure investment grows with organizational demand rather than proportional to agent count.

- Dynamic scaling on shared components contracts resources when demand is low, which is hard to do when each agent runs its own isolated stack.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The infrastructure agents need is more repetitive than the work they do. Authentication, caching, queuing, connection pooling, and cross-agent retrieval all look the same no matter which agent is calling them. When every agent provisions its own copy of this plumbing, the organization pays for the same infrastructure N times and optimizes it one team at a time. Consolidating into shared layers reverses this. Infrastructure is optimized once and every caller benefits.

A shared cache that agents call directly by host name creates tight coupling. Swapping the implementation means updating every caller. Exposing shared services through [Amazon Bedrock AgentCore Gateway](#) MCP server capabilities puts a stable interface in front of the implementation. The cache tier, queue backend, or connection pool can change without the agents noticing. [Amazon Bedrock AgentCore Identity](#) centralizes authentication so individual agents don't manage credentials independently, which is the simplest form of shared infrastructure with immediate return.

For caching specifically, the implementation choice depends on the data pattern. [Amazon ElastiCache](#) fits general-purpose hot data with flexible access patterns, and [Amazon DynamoDB Accelerator \(DAX\)](#) fits DynamoDB-backed agent state that needs microsecond reads without a separate cache layer. Both are shared across agents once provisioned. [Amazon Bedrock cross-region inference](#) distributes foundation model requests across Regions so availability is shared at the inference tier, not just at the application tier.

Deploy the agents themselves on [Amazon Bedrock AgentCore Runtime](#). Its serverless model means there is no infrastructure footprint for each agent to consolidate in the first place, which complements the shared-services pattern on the supporting tier. [Amazon Bedrock AgentCore Observability](#) exposes cache hit rates, queue depth, and invocation frequency for each shared service, so utilization data drives scaling decisions rather than theoretical peak estimates.

Implementation steps

1. **Identify common infrastructure needs:** List the plumbing duplicated across current deployments and consolidate each into a shared service layer:

- Connection pools
 - Caches
 - Queues
 - Authentication
2. **Deploy shared caching:** Provision [Amazon ElastiCache](#) for general-purpose hot data or [Amazon DynamoDB Accelerator \(DAX\)](#) for DynamoDB-backed state, so frequently accessed data is read from one cache rather than refetched per agent.
 3. **Expose shared services through a stable interface:** Publish shared infrastructure as MCP tools through [Amazon Bedrock AgentCore Gateway](#) so agents consume it without coupling to the implementation.
 4. **Centralize authentication:** Use [Amazon Bedrock AgentCore Identity](#) to manage credentials once rather than having every agent manage its own.
 5. **Distribute model inference:** Turn on [Amazon Bedrock cross-region inference](#) so foundation model capacity is pooled across Regions for availability.
 6. **Track utilization and scale on data:** Monitor the following through [Amazon Bedrock AgentCore Observability](#), and adjust capacity based on observed usage rather than worst-case estimates:
 - Cache hit rates
 - Queue depth
 - Invocation patterns

Resources

Related best practices:

- [AGENTSUS01-BP01 Design specialized agents with explicit resource boundaries](#)
- [AGENTSUS01-BP02 Implement reusable workflow patterns](#)
- [SUS03-BP02 Remove or refactor workload components with low or no use](#)

Related documents:

- [Amazon Bedrock AgentCore Gateway](#)
- [Amazon ElastiCache best practices](#)
- [Amazon DynamoDB Accelerator \(DAX\)](#)

- [Amazon Bedrock cross-region inference](#)
- [Strands Agents Tools](#)

Related examples:

- [GitHub: aws-samples/serverless-patterns](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon ElastiCache](#)
- [Amazon DynamoDB](#)

AGENTSUS01-BP04 Scale cognitive processing pathways appropriately

Foundation model inference is the single most energy-intensive operation in an agent workflow, and it runs hundreds or thousands of times a day. Matching model size, retrieval depth, and memory scope to actual task complexity keeps cognitive resource consumption proportional to the value delivered, rather than defaulting every call to the largest available model.

Desired outcome:

- You have tiered model routing in place, so each task goes to the smallest model that meets its quality bar.
- Retrieval depth and context window size are scoped to task complexity, so routine tasks don't carry the retrieval overhead of complex reasoning.
- Multimodal extraction uses purpose-built services where applicable, not raw vision models for every document.
- Agents operate within token budgets and rate limits enforced at the runtime layer for each agent.

Common anti-patterns:

- Routing every request to the largest foundation model without checking whether a smaller model or cached response would meet the quality bar, which is the largest single opportunity for energy reduction.

- Allowing agents to call models without token budgets or concurrency limits, enabling single agents to consume disproportionate resources under load.
- Configuring retrieval-augmented generation to return the same context depth for every task regardless of complexity, producing oversized context windows and redundant vector queries.
- Sending raw document images to large vision models when a purpose-built extraction service would return the same structured data at a fraction of the compute cost.

Benefits of establishing this best practice:

- Cognitive resource consumption scales with task demand rather than agent count, so the energy cost of scaling up agent fleets stays proportional to the work they do.
- Token budgets for each agent help prevent one agent from starving the rest of the fleet under load.
- Right-sizing across hundreds of daily model calls compounds into substantial energy savings that are not visible on a single-call basis.

Level of risk exposed if this best practice is not established: High

Implementation guidance

The Performance Efficiency pillar covers tiered model selection in [AGENTPERF02-BP02 Implement task-appropriate model selection strategies](#). The Cost Optimization pillar covers model cascading in [AGENCOST02-BP01 Architect tiered model selection for cost-performance optimization](#). The sustainability view adds one thing. The objective isn't latency or cost alone, but total energy and compute footprint per unit of business value delivered. A task taxonomy that ranks requests by reasoning complexity, then routes them to appropriately sized [Amazon Bedrock](#) models, makes the routing data-driven rather than default-to-biggest.

Tracking successful task completions divided by total compute consumed gives a better signal than either metric alone. A workflow that gets the right answer on the first try with a small model is more sustainable than one that uses the largest model and still retries. Tag invocations so this ratio can be calculated per task category, and use it to shift routing thresholds over time. With [Amazon Bedrock cross-region inference](#), you can distribute non-urgent requests to Regions with favorable energy profiles when latency constraints permit.

Retrieval depth in [Amazon Bedrock Knowledge Bases](#) should be a parameter of the task, not a constant. A routine question with a bounded answer doesn't need the same retrieval fanout as a

complex reasoning task. Oversized retrieval wastes vector queries and bloats context windows. For document-heavy workloads, [Amazon Bedrock Data Automation](#) extracts structured data from documents at a fraction of the compute cost of routing raw images through a vision model. The cheaper path is often the better one.

Configure AgentCore Memory with tiered TTLs and automated pruning so working memory doesn't grow unboundedly, and add semantic caching so similar queries serve cached responses instead of repeated invocations. Enforce token budgets and concurrency limits for each agent through AgentCore Runtime execution constraints. Measure actual consumption through [Amazon Bedrock AgentCore Observability](#) so thresholds stay tied to observed reality.

Implementation steps

- 1. Implement tiered model routing:** Follow the patterns in [AGENTPERF02-BP02 Implement task-appropriate model selection strategies](#) and [AGENCYCOST02-BP01 Architect tiered model selection for cost-performance optimization](#) to direct tasks to appropriately sized [Amazon Bedrock](#) models based on a complexity taxonomy.
- 2. Scope retrieval depth to task complexity:** Parameterize [Amazon Bedrock Knowledge Bases](#) retrieval so vector queries and context tokens scale with the work. Use tighter limits for routine tasks and broader retrieval only for complex reasoning.
- 3. Route document extraction to purpose-built services:** For multimodal tasks, use [Amazon Bedrock Data Automation](#) instead of sending raw images through large vision models.
- 4. Apply memory lifecycle policies:** Configure AgentCore Memory with tiered TTLs and automated pruning so working memory stays bounded and stale entries are removed automatically.
- 5. Enforce budgets and track efficiency:** Set token budgets and rate limits for each agent through AgentCore Runtime execution constraints, and track successful completions per unit of compute consumed through [Amazon Bedrock AgentCore Observability](#) to adjust routing thresholds from data.

Resources

Related best practices:

- [AGENTPERF02-BP02 Implement task-appropriate model selection strategies](#)
- [AGENCYCOST02-BP01 Architect tiered model selection for cost-performance optimization](#)
- [AGENTSUS01-BP01 Design specialized agents with explicit resource boundaries](#)

- [SUS02-BP02 Align SLAs with sustainability goals](#)

Related documents:

- [Amazon Bedrock model support](#)
- [Amazon Bedrock Knowledge Bases](#)
- [Amazon Bedrock Data Automation](#)
- [Effective cost optimization strategies for Amazon Bedrock](#)
- [Build trustworthy AI agents with Amazon Bedrock AgentCore Observability](#)
- [Agentic AI patterns and workflows on AWS - Routing dynamic dispatch patterns](#)

Related videos:

- [AWS re:Invent 2024 - Sustainable and cost-efficient generative AI with agentic workflows \(AIM333\)](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Evaluations tutorials](#)

Related services:

- [Amazon Bedrock](#)
- [Amazon Bedrock AgentCore](#)

AGENTSUS01-BP05 Adopt specification-driven tasks for frontier agents and long-running workflows

Long-running agents without explicit success criteria and resource budgets drift into exploration that never quite terminates, consuming compute hours on paths that don't deliver value. Specifications that declare acceptable outputs, cost ceilings, and termination conditions up front make extended execution a bounded investment rather than an open-ended one.

Desired outcome:

- You have specifications for each frontier agent declaring maximum execution duration, token budget, memory allocation, and termination triggers before deployment.
- Long-running workflows pause at defined checkpoints to evaluate progress against the specification, and decisions about continuing, modifying, or terminating are informed by that evaluation.
- Parent frontier agents cascade remaining budget and time to child agents they spawn, so delegation inherits the parent's ceiling.
- Specification compliance is monitored in production and feeds back into template refinement for future frontier workflows.

Common anti-patterns:

- Deploying long-running agents without explicit resource budgets or termination conditions, so unbounded exploration is structurally possible.
- Omitting success criteria and decision-making boundaries from frontier workflow configuration, producing wasteful execution patterns where compute is consumed without commensurate value.
- Running extended workflows without checkpoint-based evaluation, so nobody can make an informed decision about modifying or terminating a run in progress.
- Spawning child agents from frontier workflows without passing remaining budget downstream, breaking the cost ceiling at the first delegation.

Benefits of establishing this best practice:

- Extended workflows come with accountability, compute investment is matched against business value generated rather than consumed open-endedly.
- Checkpoint evaluations give operators a chance to redirect or halt work before it burns disproportionate infrastructure.
- Specification templates accumulate institutional knowledge about expected behavior for common frontier workload patterns.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Frontier agents, code writers, deep research agents, and autonomous planners, are open-ended by design. That makes them the place where resource discipline matters most, because the cost of an unbounded exploration is many orders of magnitude higher than the cost of a single misrouted call. A specification written before deployment gives the agent something to terminate against:

- A success criterion that says "this is done"
- A token budget that caps total consumption
- A duration limit that helps prevent indefinite runs
- Explicit termination triggers for conditions where continuation has become pointless

Without that contract, the agent runs until it happens to produce something or hits an infrastructure-imposed ceiling.

Budgets must cascade for the specification to hold. When a frontier parent delegates to child agents, the parent's ceiling becomes meaningless unless the children inherit it. This is an orchestration concern, not a platform feature. In [AWS Step Functions](#), remaining budget is passed as parameters into child workflow invocations. In a Strands-orchestrated system, it is included in the child agent's system prompt or invocation context. Make cascading explicit in the design rather than assuming children inherit by convention.

Structure long-running workflows to pause at defined points, after plan generation, after information gathering, and after each major phase, and evaluate whether progress to date justifies continued investment. Persist checkpoint state in AgentCore Memory so the evaluation can pause and resume the agent without restarting it, which preserves the sunk cost of work already done. Define the specifications themselves through [Strands Agents](#) as first-class configuration delivered at invocation time through [Amazon Bedrock AgentCore Runtime](#). This way the contract travels with the agent. Spec-driven development tools like [Kiro](#) apply the same pattern to code-writing agents, giving them directed and bounded instructions instead of open interpretation.

[Amazon Bedrock AgentCore Observability](#) tracks how often specifications are violated, what checkpoints are producing useful decisions, and where budgets are binding. Specifications that never bind and workflows that always pass checkpoints signal templates that should be loosened. Specifications that often bind signal workloads that need tighter scoping or smaller sub-agents.

Implementation steps

1. **Write specifications before deployment:** For each frontier agent, declare the following and record the success criteria that define done:
 - Maximum execution duration
 - Token budget
 - Memory allocation
 - Termination triggers
2. **Deliver specifications at invocation time:** Pass specifications into [Amazon Bedrock AgentCore Runtime](#) through [Strands Agents](#) as first-class configuration, so the contract is part of the invocation rather than implicit in the agent code.
3. **Cascade budgets to child agents:** When a parent frontier agent delegates, pass the remaining duration, token budget, and memory budget into the child invocation through [AWS Step Functions](#) parameters or Strands orchestration, so the ceiling holds across delegation.
4. **Implement checkpoint-based evaluation:** Structure long-running workflows to pause at defined points, evaluate progress against the specification, and persist state in AgentCore Memory so the agent resumes from the checkpoint after a continue/modify/terminate decision.
5. **Monitor adherence and refine templates:** Track the following through [Amazon Bedrock AgentCore Observability](#), and feed the data back into specification templates for common frontier workload patterns:
 - Specification violations
 - Checkpoint outcomes
 - Budget utilization

Resources

Related best practices:

- [AGENTSUS01-BP01 Design specialized agents with explicit resource boundaries](#)
- [AGENTSUS01-BP04 Scale cognitive processing pathways appropriately](#)
- [SUS02-BP03 Stop the creation and maintenance of unused assets](#)
- [COST02-BP02 Implement goals and targets](#)

Related documents:

- [AWS Frontier Agents](#)
- [Introducing AWS Frontier Agents and Kiro](#)
- [AWS Step Functions - Standard workflows](#)
- [Strands Agents](#)

Related examples:

- [Build multi-step applications and AI workflows with AWS Lambda durable functions](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [AWS Step Functions](#)

Resource right-sizing

Agents that right-size their dependencies (memory, caching, compute, and networking) deliver automation value without placing unsustainable load on the systems they depend on. Increased agent traffic increases the load on these dependencies. Sustainable frameworks help dependent systems scale proportionally to the benefit achieved through agentic automation.

AGENTSUS02: How do I establish sustainable frameworks for agent dependencies?

Capability intent

- Memory and context infrastructure scales with actual contextual needs rather than worst-case estimates, with tiered storage and shared persistent context separating hot and cold access paths.
- Caching is applied at every integration point, and shared caches amortize work across the agent fleet rather than being rediscovered by each agent in isolation.
- Compute, networking, and storage scale dynamically with bursty agent workloads, contracting during quiet periods and expanding during peaks without manual intervention.
- Regional and connectivity choices are deliberate, so traffic stays close to the services agents depend on and private paths are used where security or latency justifies them.

- Environmental impact is measured alongside operational metrics, and deferrable workloads are scheduled and placed with carbon-awareness to reduce footprint without affecting user-facing performance.

Maturity levels

These levels summarize what each stage of maturity looks like for resource right-sizing as a whole.

Level	Name	What it looks like
1	Initial	Agent memory, caching, and infrastructure run without deliberate right-sizing. Memory grows unboundedly, caches are isolated to each agent if they exist at all, and compute is statically provisioned for peak demand. Environmental impact isn't measured, and Region placement is treated as an afterthought.
2	Emerging	Teams have adopted Amazon Bedrock AgentCore Memory with basic retention policies and have enabled Amazon Bedrock prompt caching for stable system prompts. Amazon Bedrock AgentCore Runtime is used for some agents, and AWS Sustainability has been turned on for reporting but has not informed design decisions.

Level	Name	What it looks like
3	Defined	Tiered memory with shared persistent context through AgentCore Memory namespaces is the default for multi-agent systems. Shared caches exposed through Amazon Bedrock AgentCore Gateway amortize work across the fleet. Token caching in Amazon Bedrock AgentCore Identity reduces repeated credential validation. Amazon Bedrock AgentCore Observability tracks memory access patterns, cache hit rates, and efficiency metrics, and a 30-day environmental baseline has been established.

Level	Name	What it looks like
4	Proactive	Infrastructure scaling, Regional placement, and private connectivity through VPC interface endpoints are driven by observed patterns. Context compression, semantic retrieval through Amazon Bedrock Knowledge Bases , and streaming responses are standard. Deferrable workloads run off-peak, and Amazon Bedrock cross-region inference routes batch workloads to Regions with favorable energy profiles where latency allows. Sustainability dashboards combining operational and carbon metrics are reviewed on a regular cadence.

Level	Name	What it looks like
5	Optimized	Memory tiers, cache policies, and infrastructure provisioning are continuously recalibrated from telemetry rather than through periodic review. Environmental footprint is a first-class design input for every new agent workload, and efficiency improvements compound week over week as coverage expands. The organization contributes reasoning-cost and sustainability patterns back to its communities of practice.

Common issues to watch for

- Teams build flat memory architectures without tiered storage, so frequently accessed context competes with archival data for the same tier and retrieval costs grow with every new session.
- Organizations treat each agent instance as an isolated system with its own cache, missing the fleet-wide amortization that turns one cache hit into savings for every agent that asks the same question.
- Infrastructure stays statically provisioned for worst-case demand, so utilization sits low during normal operations and dynamic contraction never happens.
- Regional placement ignores where the frequently accessed services live, so agent traffic crosses Regions unnecessarily and adds both latency and data transfer to every invocation.
- Sustainability claims are made without baselines or trend tracking, so optimization work can't be validated and deferrable workloads still run during peak hours alongside user-facing traffic.

Best practices

- [AGENTSUS02-BP01 Optimize context management and memory utilization](#)
- [AGENTSUS02-BP02 Establish efficient agent caching strategies](#)

- [AGENTSUS02-BP03 Appropriately scale data, networking, and compute dependencies](#)
- [AGENTSUS02-BP04 Measure and optimize the environmental footprint of agent workloads](#)

AGENTSUS02-BP01 Optimize context management and memory utilization

Agent memory that grows without bounds forces every retrieval to search through increasingly large stores and every turn to reprocess history the agent has already seen. Tiered memory with retention policies keeps infrastructure scaled to the context that actually matters, so memory operations stay fast and memory cost stays proportional to use.

Desired outcome:

- You have tiered memory with active session context separated from archival data, so hot and cold tiers are not competing for the same storage.
- Retention, archival, and pruning policies keep memory bounded and automatically move aging context to the appropriate tier.
- Multi-agent systems share persistent context through namespaces rather than duplicating it per agent.
- Agents incrementally build context rather than reprocessing full interaction histories on every turn.

Common anti-patterns:

- Implementing flat memory without tiering, so hot session context competes with archival data for the same storage and access path.
- Skipping retention, archival, and pruning policies, letting memory accumulate indefinitely and forcing each retrieval to scan a larger and larger store.
- Reprocessing complete historical context on every turn instead of pulling only the relevant slice, producing redundant retrieval operations that don't improve response quality.
- Duplicating shared context across each agent's memory store rather than reading from a shared namespace, producing linear storage growth with agent count.

Benefits of establishing this best practice:

- Memory infrastructure scales with the context that actually matters, not with cumulative history.
- Semantic retrieval returns the relevant slice of context in constant time rather than scaling with store size.
- Multi-agent systems share storage for common context, reducing duplicated memory across the fleet.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Not all agent context is equally active. A recent turn in an ongoing session behaves very differently from a transcript from three months ago. The first needs millisecond access on every turn, and the second needs availability for occasional retrieval. [Amazon Bedrock AgentCore Memory](#) provides built-in tiering that separates these cases, with working memory optimized for active sessions and long-term storage for older context. Retention policies move context between tiers automatically, so the hot tier stays small and the cold tier stays cheap.

Shared persistent context is a part of the architecture where many multi-agent systems fail. When five agents each maintain their own copy of the same reference material, storage grows five times faster than the organizational demand warrants. AgentCore Memory's namespace-based organization lets multiple agents read from and write to common namespaces scoped by IAM policies, one store and many readers. This is shared storage, not shared in-process memory, so consistency and access control stay explicit. Separately, [Amazon Bedrock Knowledge Bases](#) is the right home for organizational knowledge, FAQs, and reference documentation that agents query to enrich context. It complements AgentCore Memory rather than replaces it.

Sending the full interaction history into every model call looks correct but pays to reprocess information the model already saw.

The better pattern is incremental. Maintain stateful sessions that preserve working context across turns, summarize older segments into compact representations when they age out of the immediate window, and use semantic search through Knowledge Bases with vector embeddings to retrieve only the relevant slice of history rather than the whole transcript. [Amazon Bedrock AgentCore Observability](#) exposes which memory operations are frequent and which tiers are under- or over-utilized, so allocation stays grounded in actual usage.

Implementation steps

1. **Configure tiered memory with lifecycle policies:** Use [Amazon Bedrock AgentCore Memory](#) for active session context with retention policies that automatically move aging context to long-term storage.
2. **Set up shared namespaces for multi-agent context:** Create AgentCore Memory namespaces that multiple agents read from and write to, scoped by IAM policies, so shared persistent context is stored once rather than duplicated per agent.
3. **Use semantic retrieval for historical context:** Configure [Amazon Bedrock Knowledge Bases](#) with vector embeddings so queries retrieve only the relevant slice of historical context rather than full transcripts.
4. **Compress aging context:** Apply summarization using Amazon Bedrock foundation models to condense older interaction segments into compact representations that preserve meaning at a fraction of the token cost.
5. **Monitor memory access patterns and rebalance tiers:** Track tier hit rates, retrieval latency, and store size through [Amazon Bedrock AgentCore Observability](#) and adjust retention windows and tier allocations based on observed usage.

Resources

Related best practices:

- [AGENTSUS02-BP02 Establish efficient agent caching strategies](#)
- [AGENTSUS02-BP03 Appropriately scale data, networking, and compute dependencies](#)
- [SUS02-BP01 Scale workload infrastructure dynamically](#)

Related documents:

- [Amazon Bedrock AgentCore Memory](#)
- [Amazon Bedrock AgentCore Memory: Building context-aware agents](#)
- [Building smarter AI agents: AgentCore long-term memory deep dive](#)
- [Amazon Bedrock Knowledge Bases](#)
- [Amazon Bedrock AgentCore Observability](#)

Related videos:

- [AWS 2025 - AgentCore Deep Dive: Memory](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Memory tutorials](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)

AGENTSUS02-BP02 Establish efficient agent caching strategies

Every duplicate model call, tool invocation, and memory lookup is work the agent fleet has already done once. Caching at each integration point turns repeated work into a one-time cost amortized across every caller, so resource efficiency improves as usage patterns stabilize rather than scaling linearly with traffic.

Desired outcome:

- You have caching applied at each integration point, prompt prefixes, tool results, memory lookups, and credential validation, with TTLs matched to data volatility.
- Cache layers are shared across the agent fleet so one agent's cached result benefits every other agent.
- Cache hit rates are tracked per integration point and improve as usage patterns stabilize.
- Invalidation policies help prevent stale responses where data volatility demands freshness.

Common anti-patterns:

- Making repeated calls to the same foundation model with the same stable prompt prefix instead of caching it, paying to reprocess the same tokens on every invocation.
- Caching tool results and model responses without invalidation or TTL policies, producing stale answers that appear fresh.
- Running caches isolated to each agent that don't share across the fleet, so each agent has to re-warm its own cache rather than benefiting from cached results elsewhere.

- Skipping cache instrumentation, so nobody knows which integration points have low hit rates and would benefit from a different caching strategy.

Benefits of establishing this best practice:

- Redundant processing, API calls, and network traffic are reduced at each integration point, so infrastructure cost grows sublinearly with agent usage.
- Shared cache layers mean adding agents to the fleet increases cache hit rate rather than cache pressure.
- Resource efficiency compounds over time as cache hit rates climb toward their steady-state maximum.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The sustainability view of prompt caching adds one additional consideration to this lens' existing performance and cost perspectives. The value of caching isn't just latency or cost for each call. It's cumulative compute and energy footprint across the lifetime of the fleet, and the way caching is shared determines how that cumulative footprint grows.

Caches isolated to each agent don't compound. If five agents each maintain their own cache, the fleet warms five caches instead of one, and cross-agent hits never happen. Shared cache layers exposed through [Amazon Bedrock AgentCore Gateway](#) MCP server capabilities reverse this. Every agent reads from and writes to the same cache tier, so one agent's tool result becomes the next agent's cache hit. The same principle applies at the authentication layer. [Amazon Bedrock AgentCore Identity](#) caches tokens so credential validation happens once per session across the fleet rather than once for each agent.

[Amazon Bedrock prompt caching](#) pays off fastest. Stable system prompts, the long preambles that set agent behavior, can be cached at the model layer so subsequent invocations skip reprocessing the prefix. Semantic caching adds a complementary layer where similar (not identical) queries serve cached responses after an embedding-based match, which is especially valuable for the long tail of paraphrased questions users ask.

Invalidation is where caching strategies can fail. A cache TTL calibrated to daily refresh on data that actually changes hourly serves stale content. A TTL too short to matter wastes the cache's whole purpose. Pick TTLs based on how often the underlying data actually changes, and track hit rates

through [Amazon Bedrock AgentCore Observability](#) so low-performing strategies get tuned rather than left in place.

Implementation steps

1. **Enable prompt caching for stable prefixes:** Turn on [Amazon Bedrock prompt caching](#) for stable system prompts following the patterns in [AGENTPERF03-BP04 Establish efficient agent caching and data access patterns](#) and [AGENTCOST02-BP03 Leverage intelligent caching to reduce redundant model invocations](#).
2. **Share cache layers across the fleet:** Expose tool result and semantic caches through [Amazon Bedrock AgentCore Gateway](#) MCP server capabilities so every agent reads from and writes to the same store.
3. **Cache credential validation:** Use [Amazon Bedrock AgentCore Identity](#) to cache tokens so authentication overhead happens once per session rather than once per agent invocation.
4. **Set TTLs based on data volatility:** Pick invalidation policies calibrated to how often the underlying data actually changes, shorter for live operational data, longer for stable reference material.
5. **Monitor and refine hit rates:** Track cache hit rates per integration point through [Amazon Bedrock AgentCore Observability](#) and adjust strategies where hit rates are below expectations.

Resources

Related best practices:

- [AGENTPERF03-BP04 Establish efficient agent caching and data access patterns](#)
- [AGENTCOST02-BP03 Leverage intelligent caching to reduce redundant model invocations](#)
- [AGENTSUS02-BP01 Optimize context management and memory utilization](#)
- [SUS03-BP03 Optimize areas of code that consume the most time or resources](#)

Related documents:

- [Amazon Bedrock prompt caching](#)
- [Effectively use prompt caching on Amazon Bedrock](#)
- [Optimize LLM response costs and latency with effective caching](#)
- [Amazon Bedrock AgentCore Gateway](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)

AGENTSUS02-BP03 Appropriately scale data, networking, and compute dependencies

Agent workloads have a shape that general-purpose infrastructure defaults don't fit, with bursty inference, variable-length tool execution, and unpredictable multi-step reasoning. Sizing hosting, network, and storage to the observed pattern rather than a theoretical maximum keeps infrastructure proportional to agentic work.

Desired outcome:

- Agent processes run on serverless infrastructure that scales with demand instead of static provisioning for peak load.
- Private connectivity is used where security or latency requirements justify it, not by default for every workload.
- Agent infrastructure is deployed close to the services it depends on, so cross-Region data transfer is minimized.
- Streaming responses are used for user-facing interactions to reduce memory footprint and improve time-to-first-token.
- Utilization is monitored continually and provisioning tracks actual workload demand.

Common anti-patterns:

- Applying general-purpose infrastructure configurations without analyzing the bursty inference call patterns and variable tool execution durations specific to agent workloads, producing wasteful over-allocation or performance-degrading under-provisioning.
- Maintaining static provisioning regardless of demand, reducing the ability for infrastructure to contract during low-activity periods.
- Sizing for theoretical maximum scenarios instead of right-sizing against actual demand, producing low utilization during normal operations.

- Deploying agent infrastructure far from the services it depends on, producing cross-Region network traffic that adds latency and transfer cost.

Benefits of establishing this best practice:

- Infrastructure consumption tracks demand, contracting when agents are idle and expanding during peak periods.
- Energy consumption stays proportional to the work agents deliver rather than their theoretical peak capacity.
- Private connectivity and Region colocation reduce network path length and latency where it matters.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Running agents on serverless infrastructure solves most of the static-provisioning problem by design. [Amazon Bedrock AgentCore Runtime](#) right-sizes compute per invocation with session-isolated execution, so the orchestration overhead is scoped to the session that's actually running. There's no fleet of idle EC2 instances to size against worst-case demand, because the execution unit is the invocation rather than the instance. This default makes bursty workloads affordable.

Private networking is about matching the connection pattern to the workload's actual needs. Some agents process sensitive data or have latency requirements tight enough that public internet routing is the bottleneck. For those workloads, [VPC interface endpoints for Amazon Bedrock AgentCore \(AWS PrivateLink\)](#) establishes private paths that reduce latency and keep traffic off the public internet. For workloads without those requirements, PrivateLink adds operational complexity without proportional benefit. Default to the public endpoint and promote workloads to PrivateLink when security or latency demands it.

Deploying agent infrastructure in the same AWS Region as frequently accessed Amazon Bedrock endpoints and AgentCore services reduces cross-Region data transfer overhead that compounds across thousands of daily invocations. For availability, [Amazon Bedrock cross-region inference](#) distributes foundation model requests across Regions. This is complementary rather than contradictory. Use cross-Region inference for failover and burst capacity, and keep the primary data path local.

Streaming responses change the memory profile of user-facing interactions. Without streaming, the agent accumulates the full response in memory before returning it, which means peak memory is proportional to response length. With streaming, tokens flow as they're generated and memory stays bounded. Turning on streaming in AgentCore reduces footprint for long-form interactions and improves time-to-first-token for users. [Amazon Bedrock AgentCore Observability](#) exposes the utilization data that keeps provisioning tied to actual workload demand.

Implementation steps

1. **Run agent processes on serverless runtime:** Deploy to [Amazon Bedrock AgentCore Runtime](#) for automatic scaling and session isolation, so execution capacity scales with demand rather than peak estimates.
2. **Apply private connectivity where justified:** Configure [VPC interface endpoints for Amazon Bedrock AgentCore \(AWS PrivateLink\)](#) for workloads with security or latency requirements that warrant it. Default to public endpoints otherwise.
3. **Deploy in the same Region as dependencies:** Place agent infrastructure in the Region hosting the Amazon Bedrock endpoints and AgentCore services it uses most, and turn on [Amazon Bedrock cross-region inference](#) for availability.
4. **Enable streaming for user-facing responses:** Turn on AgentCore streaming so response tokens flow as they're generated, reducing memory footprint and improving time-to-first-token.
5. **Validate utilization continually:** Track infrastructure utilization through [Amazon Bedrock AgentCore Observability](#) and adjust provisioning where observed demand is meaningfully below allocation.

Resources

Related best practices:

- [AGENTSUS02-BP01 Optimize context management and memory utilization](#)
- [AGENTSUS02-BP04 Measure and optimize the environmental footprint of agent workloads](#)
- [SUS02-BP01 Scale workload infrastructure dynamically](#)

Related documents:

- [Amazon Bedrock AgentCore Runtime](#)
- [VPC interface endpoints for Amazon Bedrock AgentCore \(AWS PrivateLink\)](#)

- [Amazon Bedrock cross-region inference](#)
- [Amazon Bedrock AgentCore Observability](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)
- [AWS PrivateLink](#)

AGENTSUS02-BP04 Measure and optimize the environmental footprint of agent workloads

Without measurement, sustainability claims are aspirational and optimizations are not tracked against real-world data. Tracking the environmental footprint of agent workloads makes sustainability an engineering metric. Baselines show where effort is worth investing, and trends show whether changes are actually working.

Desired outcome:

- You have carbon emissions baselines for agent infrastructure established across a defined observation period.
- Resource efficiency metrics for each task (tokens per successful completion, compute hours per workflow, and cache hit rates) are tracked alongside business outcomes.
- Deferrable workloads are scheduled during off-peak periods or routed to Regions with favorable energy profiles.
- Operational and sustainability metrics are combined in dashboards that inform periodic optimization reviews.

Common anti-patterns:

- Claiming sustainability benefits from agent optimizations without establishing baselines, making it impossible to validate whether changes actually reduced impact.
- Treating every workload as equally time-sensitive, running batch processing and background tasks during peak hours when deferring them would reduce contention and energy consumption.

- Ignoring regional differences in energy infrastructure when selecting deployment Regions for workloads with flexible latency requirements.
- Tracking only infrastructure utilization without tying it to business outcomes, so efficiency gains in compute don't connect to value delivered.

Benefits of establishing this best practice:

- Measurable baselines make sustainability improvements verifiable instead of aspirational.
- Optimization effort flows to the workloads with the largest environmental impact, rather than being applied uniformly.
- Deferrable workloads run when infrastructure is underutilized, improving fleet-wide efficiency without affecting user-facing performance.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

[AWS Sustainability](#) provides carbon emissions tracking with service and Region breakdowns for the infrastructure side, and [Amazon Bedrock AgentCore Observability](#) adds agent-specific metrics, tokens per successful task, compute hours per workflow, and cache hit rates that tie consumption to business outcomes. Establish baselines across a 30-day observation window so that normal workload variation is included, and track trends monthly afterward so optimization work can be validated against measured change rather than claimed change.

Not every agent workload has the same latency sensitivity. User-facing interactions need low-latency responses. Batch jobs, periodic knowledge base indexing, bulk data enrichment, evaluation runs, and non-interactive research workflows can wait hours or overnight without affecting the user. Shifting deferrable work to off-peak periods reduces resource contention on shared infrastructure and takes advantage of time windows when the broader grid is cleaner. [Amazon Bedrock cross-region inference](#) extends the same principle across geographies. Batch workloads without tight latency constraints can run in Regions with favorable energy profiles rather than defaulting to the closest one.

Amazon CloudWatch dashboards that combine operational metrics with sustainability indicators make sustainability visible in the same place operators already look. Track resource utilization efficiency, waste metrics (failed or abandoned executions as a percentage of total), and peak compared to off-peak utilization rates. Incorporate these dashboards into periodic optimization

reviews so environmental impact is a standing input to engineering priorities rather than an annual afterthought.

Implementation steps

1. **Enable carbon emissions tracking:** Turn on [AWS Sustainability](#) and establish environmental baselines for agent infrastructure across a 30-day observation window.
2. **Instrument resource efficiency per task:** Configure [Amazon Bedrock AgentCore Observability](#) to track tokens per successful completion, compute hours per workflow execution, and cache hit rates.
3. **Schedule deferrable workloads off-peak:** Identify non-interactive workloads and shift them to off-peak windows:
 - Batch processing
 - Knowledge base indexing
 - Periodic AgentCore Evaluations runs
 - Bulk data enrichment
4. **Evaluate Region placement for batch work:** For workloads with flexible latency, use [Amazon Bedrock cross-region inference](#) to route requests to Regions with favorable energy profiles.
5. **Build combined dashboards and review on a cadence:** Create Amazon CloudWatch dashboards pairing operational metrics with sustainability indicators (resource utilization efficiency, waste percentage, and peak compared to off-peak utilization), and review them as part of periodic optimization cycles.

Resources

Related best practices:

- [AGENTSUS02-BP01 Optimize context management and memory utilization](#)
- [AGENTSUS02-BP03 Appropriately scale data, networking, and compute dependencies](#)
- [SUS01-BP01 Choose Region based on both business requirements and sustainability goals](#)

Related documents:

- [AWS Sustainability User Guide](#)
- [Announcing the AWS Sustainability console](#)

- [Sustainability Pillar - AWS Well-Architected Framework](#)
- [Amazon Bedrock cross-region inference](#)
- [Amazon Bedrock AgentCore Observability](#)

Related videos:

- [AWS re:Invent 2024 - Sustainable and cost-efficient generative AI with agentic workflows \(AIM333\)](#)

Related services:

- [AWS Sustainability](#)
- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)
- [Amazon CloudWatch](#)

Agent governance

Organizations that deliberately design how agents interact with users and business processes sustain both automation value and institutional expertise as adoption scales. As agents proliferate, there is a risk that internal expertise will be diluted or lost to fully automated processes. The patterns governing agent interactions must be deliberately designed and maintained, establishing reusable approaches based on real-world organizational needs.

AGENTSUS03: How do I establish durable patterns for agent interactions with users and business processes?

Capability intent

- Organizational competencies are deliberately categorized across human-owned, agent-augmented, and fully automated tiers, with clear boundaries enforced by routing and guardrail policies.
- Agent development targets processes the organization has already mastered, so automation accelerates proven workflows rather than substituting for process discovery.

- Agents ship with declarative specifications, dependency graphs, and policy documentation that remain current as agents evolve, so institutional knowledge is preserved in an accessible form.
- A central catalog of deployed agents captures owner, purpose, dependencies, and usage, and teams consult it before building new agents.
- The agent portfolio is actively rationalized, with inactive agents flagged for review, decommissioned through a structured lifecycle, and reclaimed infrastructure returned to the shared pool.

Maturity levels

These levels summarize what each stage of maturity looks like for agent governance as a whole.

Level	Name	What it looks like
1	Initial	Agents are built and deployed without competency tiering, ownership assignment, or portfolio oversight. Documentation is thin, decommissioning is one-time, and institutional knowledge is held only by the original developers rather than recorded in any specification.
2	Emerging	Teams have categorized competencies into human-owned, agent-augmented, and fully automated tiers, and Amazon Bedrock Guardrails are in place for high-stakes escalation. Declarative configuration in Amazon Bedrock AgentCore captures the baseline agent specification. Basic ownership

Level	Name	What it looks like
		metadata is recorded for each deployed agent.
3	Defined	Documented workflows feed agent development through Amazon Bedrock Knowledge Bases RAG sources, and Amazon Bedrock AgentCore Gateway routing separates routine tasks from cases that require human judgment. Amazon Bedrock AgentCore Evaluations compares agent outputs against expert baselines, and rotation programs keep subject matter experts practiced in the underlying work. A central agent catalog with ownership and purpose is maintained.

Level	Name	What it looks like
4	Proactive	Runtime documentation generated through Amazon Bedrock AgentCore Observability validates design specifications against actual behavior, and production promotion is gated on current documentation. Amazon CloudWatch metrics flag inactive agents for owner review, and a structured decommissioning lifecycle moves agents through active, under review, deprecated, and decommissioned states. Quarterly portfolio rationalization reviews are a standing cadence.

Level	Name	What it looks like
5	Optimized	Governance is continuous rather than periodic. Portfolio health metrics (total agent count, percentage with active usage, percentage with current documentation) are reviewed at the organizational level. Specification currency, evaluation baselines, and competency tiering adjust based on operational telemetry, and the organization shares its agent governance patterns with its communities of practice.

Common issues to watch for

- Teams automate unfamiliar or poorly understood processes where decision criteria and success metrics are not documented, so agent development wastes resources on untested methodologies.
- Organizations skip competency tiering, so agents end up handling high-stakes or ambiguous decisions that should be routed to human experts and subject matter expertise quietly atrophies.
- Teams treat agent development as code-only artifacts without capturing the expert decision-making patterns and business logic rationale, leaving institutional knowledge locked inside implementations that only the original developers understand.
- Agents are deployed without ownership assignment or usage tracking, so there is no basis for evaluating whether an agent is still delivering value or has fallen out of use.
- Teams build new agents for capabilities that already exist elsewhere in the organization because no one searches the catalog first, creating redundant implementations and undermining reuse.

Best practices

- [AGENTSUS03-BP01 Maintain organizational skills and competencies](#)
- [AGENTSUS03-BP02 Build agents to mirror your organizational skills and competencies](#)
- [AGENTSUS03-BP03 Maintain comprehensive specifications for agents and agentic systems](#)
- [AGENTSUS03-BP04 Decommission unused agents and prevent agent sprawl](#)

AGENTSUS03-BP01 Maintain organizational skills and competencies

Organizations that automate without deliberately preserving the human expertise behind the automation lose the capacity to train new staff, handle edge cases, and recover when automated systems reach their limits. Keeping a clear distinction between what agents handle autonomously and what stays with human experts sustains organizational capability across the whole lifetime of the deployment.

Desired outcome:

- You have a competency taxonomy that separates human-owned, agent-augmented, and fully automated tasks, with documented criteria for each tier.
- Routing in the agent layer escalates high-stakes, ambiguous, or edge-case decisions to human experts automatically.
- Rotation programs and workshops keep subject matter experts proficient with the workflows agents otherwise execute.
- You monitor escalation rates and expert task distribution to confirm critical competencies stay actively practiced.

Common anti-patterns:

- Automating domain workflows without maintaining documented runbooks or periodic manual execution exercises, reducing the organization's capacity to operate when agent systems are unavailable.
- Running agents without clear boundaries between autonomous action and human oversight, producing cases where agents make high-stakes decisions that should have routed to experts.
- Scaling agent adoption without workforce planning to preserve critical skills, so short-term productivity gains erode long-term organizational capacity.
- Treating escalation as an exception rather than an expected outcome, so the hand-off paths between agents and human experts are undertested and fail when they are needed most.

Benefits of establishing this best practice:

- The organization retains the capacity to operate manually when agent systems are unavailable or encounter situations beyond their training.
- Critical expertise stays available for onboarding, edge cases, and evolving business processes that automation has not caught up to.
- Adoption paces the organization's ability to maintain oversight, rather than outrunning it.

Level of risk exposed if this best practice is not established: High

Implementation guidance

The threshold question is which competencies must stay with humans and which can move to agents or automation. A three-tier taxonomy, human-owned, agent-augmented, and fully automated, gives teams a shared vocabulary for that decision. The categorization criteria matter more than the labels:

- Human-owned means decisions where error tolerance is low and context is highly variable (regulatory judgment, customer escalations, and strategic trade-offs)
- Agent-augmented means work where an agent accelerates human output but the human remains the decision-maker (code review, document drafting, and data analysis)
- Fully automated means routine tasks where agent accuracy exceeds the cost of errors (routing, classification, and standard form processing)

Categorization is reviewed as agent capabilities improve and as organizational priorities shift.

[Amazon Bedrock AgentCore Gateway](#) with Policies and code interceptors can invoke human-in-the-loop workflows based on complexity thresholds, confidence scores, or stakes assessment. [Amazon Bedrock Guardrails](#) enforces the decision boundaries so that high-stakes cases escalate automatically rather than depending on the agent to self-report low confidence. [Amazon Bedrock AgentCore Evaluations](#) runs agent outputs against expert-generated baselines, which is how you detect drift in categories where the agent was trusted but is now degrading.

The automation that removes a task from expert workflows also removes the practice that kept expertise sharp. Rotation programs assign experts to handle a fraction of cases manually on a recurring basis. For business-critical competencies, a defined minimum (a percentage of cases per quarter, a weekly shift, or a monthly workshop) keeps practice active. Document runbooks for

workflows agents now execute so the manual path remains viable when the automated one is unavailable. [Amazon Bedrock AgentCore Observability](#) tracks escalation rates and task distribution, showing whether experts are being kept in the loop often enough to stay sharp.

Implementation steps

1. **Define a competency taxonomy:** Categorize organizational skills into human-owned, agent-augmented, and fully automated tiers with documented criteria and a review cadence.
2. **Configure automated escalation:** Use [Amazon Bedrock AgentCore Gateway](#) Policies and [Amazon Bedrock Guardrails](#) to route high-stakes, ambiguous, or low-confidence decisions to human experts.
3. **Establish rotation programs:** Assign subject matter experts to handle a defined percentage of cases manually each quarter for competencies critical to business resilience, and maintain runbooks for manual execution.
4. **Validate agent outputs against expert baselines:** Run [Amazon Bedrock AgentCore Evaluations](#) against baselines produced by subject matter experts to detect drift in categories that have been trusted to automation.
5. **Monitor escalation and task distribution:** Track escalation rates and expert task distribution through [Amazon Bedrock AgentCore Observability](#) to confirm that critical competencies remain actively practiced.

Resources

Related best practices:

- [AGENTSUS03-BP02 Build agents to mirror your organizational skills and competencies](#)
- [AGENTSUS03-BP03 Maintain comprehensive specifications for agents and agentic systems](#)
- [MLPERF06-BP01 Include human-in-the-loop monitoring](#)

Related documents:

- [Amazon Bedrock Guardrails](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [Human-in-the-loop \(HITL\) - Amazon Nova Act](#)
- [Build reliable AI agents with Amazon Bedrock AgentCore Evaluations](#)
- [Amazon Bedrock AgentCore Evaluations](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - Evaluations](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)

AGENTSUS03-BP02 Build agents to mirror your organizational skills and competencies

Agents built to codify proven workflows deliver immediate value. Agents built to automate processes the organization has not mastered itself usually waste resources learning what could have been documented first. Focusing automation on well-understood work is the difference between agent adoption that pays back quickly and adoption that consumes resources on untested methodology.

Desired outcome:

- You select processes for automation where steps, decision criteria, and success metrics are documented and consistently executed by human experts.
- Agents mirror the decision trees, validation checkpoints, and escalation paths that experts already use.
- Institutional knowledge is captured in knowledge bases that ground agent decisions, rather than relying solely on foundation model training data.
- Readiness criteria, minimum documentation maturity, gate agent development projects before resources are committed.

Common anti-patterns:

- Building agents to automate unfamiliar or poorly understood processes where steps, decision criteria, and success metrics are not documented, so the automation becomes an experiment rather than a productivity gain.
- Skipping the step of codifying expert practices before writing agent logic, missing the opportunity to replicate proven approaches.

- Generating agent code without understanding it, producing implementations the team can't maintain or evolve and accumulating technical debt.
- Automating processes that vary widely in execution across experts, so the agent picks one variant and discovers at runtime that the choice did not match the business context.

Benefits of establishing this best practice:

- Agent automation starts from proven practice, so the automation runs correctly at deployment instead of being debugged in production.
- Human expertise remains the source of truth and is amplified rather than replaced, preserving adaptation capacity.
- Development resources flow to workflows where ROI is defensible, rather than being spent on speculative automation.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

The criterion for a good automation candidate is straightforward. The process is well understood, the steps are documented, the decision criteria are explicit, and multiple experts execute it the same way. Processes that fail that test should not be the first things an organization automates. The agent will replicate whichever variant the documentation captures, and if the documentation captures the wrong variant, every agent invocation reinforces the error. The documentation exercise itself is the first return on investment. Writing down what experts do encourages consistency even before any agent is built.

Once the documentation exists, the agent is a translation of it. You configure Amazon Bedrock AgentCore agents to follow the same decision trees, validation checkpoints, and escalation paths as human experts. This gives the automation the same runtime behavior as the manual process. [Amazon Bedrock Knowledge Bases](#) holds the documented expertise as a RAG source, which grounds agent decisions in institutional knowledge rather than leaving them to foundation model training data. [Amazon Bedrock AgentCore tools](#) mirror the external systems experts use (the same APIs, the same data sources, and the same validation services), so the agent's view of the task matches the expert's.

Routing creates a distinction between automated and human-owned work. [Amazon Bedrock AgentCore Gateway](#) can direct routine tasks to full automation while escalating complex

or ambiguous work to human experts, implementing the three-tier taxonomy described in [AGENTSUS03-BP01 Maintain organizational skills and competencies](#) at runtime. [Amazon Bedrock AgentCore Evaluations](#) assesses whether agent outputs match expert-generated baselines, and [Amazon Bedrock AgentCore Observability](#) exposes patterns where agent behavior has drifted from documented practice. Gate agent development projects behind readiness criteria, inputs, outputs, decision criteria, and success metrics need to be specified before development starts, so the documentation discipline becomes a precondition rather than an afterthought.

Spec-driven development tools like Kiro apply the same discipline to the implementation side. Agent code written from a specification is more maintainable, more reviewable, and less likely to bake in assumptions no one can trace later. The tradeoff is upfront effort on the specification, which is generally recouped during review, debugging, and evolution.

Implementation steps

1. **Document workflows before automating them:** Capture the following from subject matter experts before approving agent development:
 - Decision logic
 - Exception handling
 - Success criteria
 - Readiness criteria (inputs, outputs, decision criteria, and metrics)
2. **Store institutional knowledge as RAG sources:** Load documented expertise into [Amazon Bedrock Knowledge Bases](#) so agent decisions ground in the organization's knowledge rather than foundation model defaults.
3. **Mirror expert system access:** Configure [Amazon Bedrock AgentCore tools](#) to give agents the same external systems and data sources human experts use.
4. **Route routine vs. complex work:** Use [Amazon Bedrock AgentCore Gateway](#) to send routine tasks to full automation and escalate complex or ambiguous work to human experts.
5. **Validate behavior against expert baselines:** Run [Amazon Bedrock AgentCore Evaluations](#) against expert-generated baselines and monitor drift through [Amazon Bedrock AgentCore Observability](#).

Resources

Related best practices:

- [AGENTSUS03-BP01 Maintain organizational skills and competencies](#)
- [AGENTSUS03-BP03 Maintain comprehensive specifications for agents and agentic systems](#)
- [OPS08-BP01 Use runbooks to perform procedures](#)

Related documents:

- [Amazon Bedrock Knowledge Bases](#)
- [Amazon Bedrock AgentCore tools](#)
- [Amazon Bedrock AgentCore Evaluations](#)
- [Amazon Bedrock AgentCore Gateway](#)
- [AI agents in enterprises: Best practices with Amazon Bedrock AgentCore](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon Bedrock](#)

AGENTSUS03-BP03 Maintain comprehensive specifications for agents and agentic systems

Agents without specifications become systems that only their original developers understand. Thorough documentation, purpose, boundaries, integration points, and decision logic keep institutional knowledge available to the whole team and verify that agent behavior is traceable as teams and business processes change.

Desired outcome:

- Each deployed agent has a specification covering business purpose, operational boundaries, decision criteria, and escalation paths.
- Specifications are stored in version-controlled artifacts alongside the agent implementation.
- A centralized agent catalog lets team members discover, review, and reuse agent capabilities.
- Runtime behavior is documented automatically and compared against design specifications to detect drift.
- Governance requires specification validation before agents reach production.

Common anti-patterns:

- Deploying agents without thorough documentation of purpose, boundaries, or decision-making logic, so institutional knowledge is held only by the original developers.
- Letting documentation fall out of date as agents evolve, producing specifications that describe yesterday's behavior rather than today's.
- Treating agent configuration as code only, without capturing the expert decision-making patterns and business logic rationale that informed the design.
- Skipping specification validation in the promotion path, so agents reach production with documentation that doesn't match their actual behavior.

Benefits of establishing this best practice:

- Institutional knowledge survives personnel changes and organizational restructuring, preserved in documentation rather than memory.
- Teams develop against specifications informed by business process, reducing the experimentation cost of each new agent.
- Documented decision logic and operational boundaries make oversight of automated processes tractable.

Level of risk exposed if this best practice is not established: High

Implementation guidance

A specification decouples the agent's behavior from its author. When the original developer moves on, the next maintainer picks up the system through the specification rather than through reverse engineering. Amazon Bedrock AgentCore enables declarative specifications that capture prompt templates, tool definitions, guardrail policies, and orchestration logic as version-controlled artifacts. The documentation and the configuration are the same artifact rather than two copies that drift.

Spec-driven development tools like Kiro extend the discipline into how agents get written in the first place. When specifications are the starting point rather than an afterthought, documentation is produced as a byproduct of development rather than a retrospective task. The upfront investment in writing the specification is recovered during code review, testing, and ongoing evolution, because every subsequent change happens against a clear baseline.

Discovery needs a central home, like a catalog based on [Amazon Bedrock AgentCore Registry](#) registry capabilities that makes agents discoverable, governable, and reusable across the organization. Each entry captures business purpose, version history, dependencies, and operational characteristics, so a team evaluating whether to build something new can check what already exists.

Specifications can drift if an organization lacks validation mechanisms. [Amazon Bedrock AgentCore Observability](#) generates runtime documentation from actual execution patterns. When runtime behavior diverges from design specifications, the telemetry shows it before the divergence becomes undocumented institutional knowledge.

For multi-agent systems, document coordination protocols and delegation patterns explicitly, because the emergent behavior of a multi-agent system is rarely obvious from any individual agent's specification. Portfolio-level monitoring, specification compliance, documentation currency, and behavioral drift, keeps documentation usable as the agent count scales.

Implementation steps

1. **Set documentation standards:** Require each agent to carry specifications using AgentCore's declarative configuration artifacts. Each specification must cover:
 - Business purpose
 - Operational boundaries
 - Decision criteria
 - Escalation paths
2. **Adopt spec-driven development:** Use Kiro or similar spec-driven tools so documentation is produced as a natural byproduct of the development process rather than a retrospective task.
3. **Register agents in a central catalog:** Record each agent in [Amazon Bedrock AgentCore Registry](#) with the following metadata:
 - Purpose
 - Version history
 - Dependencies
 - Operational characteristics
4. **Generate runtime documentation:** Use [Amazon Bedrock AgentCore Observability](#) to produce runtime documentation that can be compared against design specifications to detect drift.

5. **Gate promotion on specification validation:** Require documentation validation as part of the pipeline that promotes agents to production, so production agents always have current specifications.

Resources

Related best practices:

- [AGENTSUS03-BP01 Maintain organizational skills and competencies](#)
- [AGENTSUS03-BP04 Decommission unused agents and prevent agent sprawl](#)
- [OPS11-BP01 Have a process for continuous improvement](#)

Related documents:

- [Amazon Bedrock AgentCore Registry](#)
- [The future of managing agents at scale: AWS Agent Registry now in preview](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Amazon Bedrock Agents Documentation](#)

Related examples:

- [GitHub: awslabs/amazon-bedrock-agentcore-samples - End-to-end use cases](#)

Related services:

- [Amazon Bedrock AgentCore](#)

AGENTSUS03-BP04 Decommission unused agents and prevent agent sprawl

Every agent that stays deployed past its usefulness consumes infrastructure, expands the attack surface, and adds operational overhead that never shows up as an explicit line item. Active portfolio management helps prevent the silent accumulation of cost and complexity that comes with scaled adoption.

Desired outcome:

- Every deployed agent has a documented owner, a clear business purpose, and measurable usage.
- Agents that no longer deliver value move through a structured decommissioning lifecycle and are retired.
- Teams search the agent registry for existing capabilities before initiating new agent development.
- Portfolio health, total agent count, percentage with active usage, percentage with current documentation, is visible at the organizational level.

Common anti-patterns:

- Deploying agents without ownership assignment or usage tracking, so no one can tell which agents still deliver value.
- Allowing abandoned agents to persist indefinitely because no decommissioning process exists, accumulating infrastructure cost and expanding the attack surface.
- Building new agents for capabilities that already exist in deployed agents elsewhere in the organization, creating redundant implementations.
- Relying on informal knowledge of which agents are still useful instead of automated usage tracking, so the decommissioning decision depends on whoever happens to remember which agents are deployed.

Benefits of establishing this best practice:

- Portfolio-level decisions about resource consumption are tied to actual business value delivered, rather than historical deployment patterns.
- Decommissioning reclaims infrastructure resources and reduces the operational surface area.
- Discoverability of existing capabilities reduces redundant implementations and reinforces reusable architecture patterns.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

[Amazon Bedrock AgentCore Registry](#) provides the authoritative catalog of deployed agents, with each entry capturing business purpose, designated owner, deployment date, dependencies, and usage metrics. Semantic capability search lets teams discover existing agents before building

new ones, which is the preventive side of avoiding agent sprawl. Pair the registry with [Amazon CloudWatch](#) metrics tracking invocation frequency and last-invocation timestamp for each agent. Inactive agents then surface automatically for owner review instead of being discovered during audits.

Decommissioning becomes routine when it has a defined lifecycle. Active, under review, deprecated, and decommissioned stages give owners clear transitions and give the organization consistent visibility into which agents are on the path to retirement. When an agent is flagged for low usage, the owner's first question is whether it serves a seasonal or infrequent-but-critical purpose. Tax-season agents, quarterly reporting agents, and disaster-recovery agents appear idle most of the time but are essential when they are invoked. A structured review makes that distinction before deprecation happens.

During quarterly portfolio rationalization, teams evaluate the full agent inventory against current business priorities, identify overlapping capabilities, and merge redundant implementations into shared patterns (following [AGENTSUS01-BP02 Implement reusable workflow patterns](#)). They retire agents whose business context has changed. Portfolio health metrics tracked through [Amazon Bedrock AgentCore Observability](#), total agent count, percentage with active usage, and percentage with current documentation, make sustainability outcomes visible at the organizational level.

Implementation steps

1. **Register every deployed agent with ownership metadata:** Record each agent in [Amazon Bedrock AgentCore Registry](#) with the following:
 - Business purpose
 - Designated owner
 - Deployment date
 - Dependencies
2. **Track invocation metrics and flag inactive agents:** Instrument agents with [Amazon CloudWatch](#) metrics capturing invocation frequency and last-invocation timestamp, and flag agents that cross a defined inactivity threshold for owner review.
3. **Define a decommissioning lifecycle:** Establish the following stages with transition criteria and owner responsibilities at each:
 - Active
 - Under review
 - Deprecated

- Decommissioned
4. **Require registry search before new agent development:** Add a pre-development check to the intake process so teams discover and evaluate existing capabilities before initiating new work.
 5. **Run quarterly portfolio reviews:** Evaluate the full agent inventory against current business priorities, consolidate overlapping capabilities into shared patterns, and retire agents whose business context has changed.

Resources

Related best practices:

- [AGENTSUS01-BP02 Implement reusable workflow patterns](#)
- [AGENTSUS03-BP03 Maintain comprehensive specifications for agents and agentic systems](#)
- [AGENTSUS02-BP04 Measure and optimize the environmental footprint of agent workloads](#)
- [SUS03-BP02 Remove or refactor workload components with low or no use](#)

Related documents:

- [Amazon Bedrock AgentCore Registry](#)
- [The future of managing agents at scale: AWS Agent Registry now in preview](#)
- [Amazon Bedrock AgentCore Observability](#)
- [Amazon CloudWatch metrics](#)

Related videos:

- [AgentCore Registry: Discover, Govern, and Reuse AI Agents at Scale](#)

Related services:

- [Amazon Bedrock AgentCore](#)
- [Amazon CloudWatch](#)

Additional resources

- [AWS Sustainability](#): Track, measure, and report carbon emissions associated with AWS usage.
- [Announcing the AWS Sustainability console](#)
- [Announcing the Updated AWS Well-Architected Generative AI Lens](#)

Conclusion

The Agentic AI Lens provides a thorough framework for building agentic AI systems that are operationally excellent, secure, reliable, high-performing, cost-effective, and sustainable. As agentic AI transitions from experimental technology to production workload, the architectural decisions made during system design have lasting consequences for the reliability, security, and cost profile of deployed systems.

Key themes

Several themes emerge consistently across the pillars of this lens:

- **Modularity enables reliability and efficiency:** Agentic systems built from specialized, single-purpose agents with clear interfaces are more straightforward to test, debug, scale, and optimize than monolithic agent architectures. The actor model pattern, where each agent encapsulates a single capability and communicates exclusively through message passing, provides a proven foundation for building reliable, maintainable agentic environments.
- **Observability is foundational, not optional:** The stochastic nature of LLM-powered agents means that unexpected behavior is inevitable. Systems that lack thorough observability, distributed tracing, structured logging, behavioral baselines, and anomaly detection, can't detect or diagnose failures before they cascade. Observability must be designed into agentic systems from inception, not added as an afterthought.
- **Graceful degradation preserves value during failures:** Agentic systems that fail completely when any component degrades provide less value than systems that maintain partial functionality with transparent communication about reduced capabilities. Every critical dependency in an agentic system must have a defined fallback behavior that activates automatically when the primary path is unavailable.
- **Human oversight remains essential:** Even highly capable agentic systems benefit from human oversight at appropriate decision points. Tiered oversight models, where low-risk actions proceed autonomously, medium-risk actions trigger notifications, and high-risk actions require explicit approval, balance operational efficiency with appropriate governance. The goal isn't to remove human judgment but to apply it where it matters most.
- **Cost and sustainability require intentional design:** The token-based cost model of LLM inference, combined with the iterative reasoning loops of agentic systems, creates cost dynamics that differ fundamentally from traditional cloud workloads. Cost-aware design, right-sizing

models to task complexity, implementing efficient context management, and establishing full cost visibility, must be a first-class architectural concern rather than a post-deployment optimization.

The evolving field

Agentic AI is a rapidly evolving field. The models, frameworks, protocols, and services available to builders are advancing quickly, and the architectural patterns that represent best practices today will continue to evolve. This lens will be updated as the field matures and as new patterns emerge from production deployments.

The principles underlying this lens, modularity, observability, graceful degradation, human oversight, and cost awareness, are durable regardless of how the specific technologies evolve. Teams that internalize these principles will be well-positioned to adapt their architectures as the agentic AI field continues to develop.

Next steps

Teams beginning their agentic AI journey can:

1. Review the design principles to understand the foundational values that guide architectural decisions across all pillars.
2. Consult the definitions to establish a shared vocabulary for discussing agentic AI architecture.
3. Work through each pillar's focus areas and best practices, prioritizing high-risk practices for immediate implementation.
4. Use the implementation guidance and code examples in each best practice as starting points for your specific context, adapting them to your team's technology choices and operational requirements.
5. Establish regular Well-Architected reviews that use this lens to assess your agentic AI systems as they evolve.

Building reliable, secure, and cost-effective agentic AI systems is challenging work. The best practices in this lens represent the collective experience of teams who have navigated these challenges in production. We hope this guidance accelerates your journey and helps you build agentic AI systems that deliver lasting value.

Contributors

The following individuals and organizations contributed to this document.

- Stephen Salim, Well-Architected Innovation Senior Solutions Architect, Amazon Web Services
- Steven DeVries, Principal GenAI Specialist Solution Architect, Amazon Web Services
- Dan Ferguson, Senior GenAI Specialist Solution Architect, Amazon Web Services
- Rosa Elena Lorenzana, ISV SA Leader, Amazon Web Services
- Gopi Krishnamurthy, Senior AI/ML Specialist Solution Architect, Amazon Web Services
- Aaron Sempf, Next-Gen Tech Leader, Amazon Web Services
- Alastair Bate, Guidance Lead Solutions Architect, Amazon Web Services
- Haleh Najafzadeh, Senior Manager Guidance, Amazon Web Services
- Stewart Matzek, Senior Technical Writer Well-Architected, Amazon Web Services
- Anthony Harvey, Senior Security Specialist Solutions Architect, Amazon Web Services
- Bhushan Bhale, Senior Solutions Architect, Amazon Web Services
- Dustin Liukkonen, Senior Solutions Architect, Amazon Web Services
- Anand Komandooru, Principal Delivery Consultant, Amazon Web Services
- Zoran Ivanovic, Principal Delivery Consultant, Amazon Web Services
- Melanie Li, Senior AIML Specialist Solutions Architect, Amazon Web Services
- Maitreya Ranganath, Principal Security Solutions Architect, Amazon Web Services
- James Schafer, Security Solutions Architect, Amazon Web Services
- Cameron Smith, Senior Delivery Consultant, Amazon Web Services
- Chris Lamont-Smith, Senior AIML Security Specialist SA, Amazon Web Services
- Ram Ramani, Principal Security Solutions Architect, Amazon Web Services
- Riggs Goodman III, Principal Security Solutions Architect, Amazon Web Services
- Adam Weber, Senior Solutions Architect, Amazon Web Services
- Bharath Sridharan, Principal TAM, Amazon Web Services
- Huseyin Genc, Senior Solutions Architect Private Equity, Amazon Web Services
- Marcus Watson, Senior Solutions Architect Private Equity, Amazon Web Services
- Nick McCarthy, Senior WW SSA Model Customization GenAI, Amazon Web Services
- Sandeep Rohilla, Senior Solutions Architect, Amazon Web Services

- Madhuri Srinivasan, Senior Technical Writer Well-Architected, Amazon Web Services
- Nataliya Godunok, Senior Guidance Lead Solutions Architect, Amazon Web Services
- Alex Pieri, Solutions Architect, Amazon Web Services

Acknowledgments

- [AWS Well-Architected Framework](#)
- [Operationalizing agentic AI on AWS](#)
- [Foundations of agentic AI on AWS](#)
- [Agentic AI frameworks, protocols, and tools on AWS](#)
- [Agentic AI patterns and workflows on AWS](#)
- [Security for agentic AI on AWS](#)
- [Building serverless architectures for agentic AI on AWS](#)
- [Building multi-tenant architectures for agentic AI on AWS](#)
- [Economics for agentic AI on AWS](#)

Appendix A: Best practice reference

This appendix provides a complete reference of all best practices defined in the Agentic AI Lens, organized by pillar and focus area.

Operational excellence

AGENTOPS01: How do you establish operational practices for agentic AI systems?

ID	Best practice	Risk
AGENTOPS01-BP01	Establish well-defined agent roles, responsibilities, and success criteria	High
AGENTOPS01-BP02	Design multi-agent handoff procedures with human-in-the-loop escalation	High
AGENTOPS01-BP03	Develop test scenarios that accurately capture failures of dependent components, orchestration protocols, and business processes	High

AGENTOPS02: How do you manage prompt and configuration lifecycle?

ID	Best practice	Risk
AGENTOPS02-BP01	Evolve agent prompts, tool calls, and configurations to reflect evolving business needs	High

ID	Best practice	Risk
AGENTOPS02-BP02	Implement configuration drift detection and remediation	High
AGENTOPS02-BP03	Implement agent behavior versioning and rollback capabilities	High
AGENTOPS02-BP04	Maintain feedback control loops for continuous improvement	Medium

AGENTOPS03: How do you manage agent lifecycle and deployment processes?

ID	Best practice	Risk
AGENTOPS03-BP01	Define an agent lifecycle with clear SME ownership, testing, and governance	High
AGENTOPS03-BP02	Implement CI/CD pipelines tailored to agentic system deployment (AgentOps)	High
AGENTOPS03-BP03	Implement agent-specific scaling policies and capacity planning	Medium
AGENTOPS03-BP04	Implement organizational agent portfolio management and governance at scale	High

AGENTOPS04: How do you establish tool integration and management practices?

ID	Best practice	Risk
AGENTOPS04-BP01	Implement tool registry and catalog management	Medium
AGENTOPS04-BP02	Establish standardized tool integration protocols (MCP, A2A)	High
AGENTOPS04-BP03	Develop fallback behavior and error handling for tool invocations	Medium

AGENTOPS05: How do you implement comprehensive observability and monitoring for agentic systems?

ID	Best practice	Risk
AGENTOPS05-BP01	Establish end-to-end tracing and telemetry for agent operations	High
AGENTOPS05-BP02	Monitor agent behavior patterns and detect anomalies	High
AGENTOPS05-BP03	Implement structured logging and comprehensive audit trails	High
AGENTOPS05-BP04	Define and track KPIs for agent workflows	Medium

ID	Best practice	Risk
AGENTOPS05-BP05	Create workflow-specific dashboards for operational health	Medium

AGENTOPS06: How do you implement testing, evaluation, and validation frameworks?

ID	Best practice	Risk
AGENTOPS06-BP01	Design multi-layered testing frameworks	High
AGENTOPS06-BP02	Evaluate and track ongoing agent performance	High
AGENTOPS06-BP03	Establish SME-driven validation and business approval workflows	High

AGENTOPS07: How do you establish operational recovery and consumption monitoring?

ID	Best practice	Risk
AGENTOPS07-BP01	Implement automated response and recovery mechanisms	High
AGENTOPS07-BP02	Establish operational knowledge management systems	Medium

ID	Best practice	Risk
AGENTOPS07-BP03	Augment change management to accommodate technical improvements and business requirements	Medium
AGENTOPS07-BP04	Implement break-glass operational runbooks	High

Security

AGENTSEC01: How do you secure agentic memory and securely manage state between agents?

ID	Best practice	Risk
AGENTSEC01-BP01	Implement memory isolation and integrity controls	High
AGENTSEC01-BP02	Validate and sanitize memory inputs	High
AGENTSEC01-BP03	Monitor for hallucination propagation	Medium

AGENTSEC02: How do you control and secure agent tool usage?

ID	Best practice	Risk
AGENTSEC02-BP01	Implement tool authorization	High
AGENTSEC02-BP02	Validate tool inputs and outputs	High

ID	Best practice	Risk
AGENTSEC02-BP03	Maintain approved tool registry with security assessments	Medium

AGENTSEC03: How do you manage agent identities, permissions, and prevent privilege escalation?

ID	Best practice	Risk
AGENTSEC03-BP01	Implement strong authentication for agent identities	High
AGENTSEC03-BP02	Separate agent and human user permission	High
AGENTSEC03-BP03	Implement least privilege with dynamic boundaries	High
AGENTSEC03-BP04	Regular permission audits and access reviews	Medium

AGENTSEC04: How do you support agent goal alignment and prevent manipulation?

ID	Best practice	Risk
AGENTSEC04-BP01	Implement guardrails and alignment controls	High
AGENTSEC04-BP02	Human-in-the-loop for critical decisions	High

AGENTSEC05: How do you implement observability and prevent repudiation?

ID	Best practice	Risk
AGENTSEC05-BP01	Implement comprehensive logging and decision artifact storage	High
AGENTSEC05-BP02	Implement distributed tracing for agent interactions	Medium

AGENTSEC06: How do you secure multi-agent orchestration and coordination?

ID	Best practice	Risk
AGENTSEC06-BP01	Encrypt and sign inter-agent messages	High
AGENTSEC06-BP02	Implement workflow orchestration security controls	High
AGENTSEC06-BP03	Establish trust boundaries between agents	High
AGENTSEC06-BP04	Monitor and detect coordination anomalies	Medium

AGENTSEC07: How do you protect human oversight from manipulation and detect rogue agents?

ID	Best practice	Risk
AGENTSEC07-BP01	Implement cognitive load management	Medium
AGENTSEC07-BP02	Clear confidence indicators and manipulation warnings	Medium
AGENTSEC07-BP03	Multiple reviewers for critical operations	Medium
AGENTSEC07-BP04	Behavioral anomaly detection and agent containment	High
AGENTSEC07-BP05	Regular security assessments and red teaming	Medium

AGENTSEC08: How do you validate and secure agent inputs and outputs?

ID	Best practice	Risk
AGENTSEC08-BP01	Multi-layer input validation and prompt injection defense	High
AGENTSEC08-BP02	Output filtering for sensitive information	High

AGENTSEC09: How do you perform vulnerability scanning and penetration testing for agentic AI systems?

ID	Best practice	Risk
AGENTSEC09-BP01	Integrate AI-powered vulnerability scanning across the development lifecycle	High
AGENTSEC09-BP02	Conduct context-aware penetration testing with multi-agent attack simulation	High
AGENTSEC09-BP03	Implement continuous security validation with automated remediation	Medium
AGENTSEC09-BP04	Establish scoped and controlled testing environments for agent security assessments	Medium
AGENTSEC09-BP05	Implement runtime threat detection, security event correlation, and automated remediation for agents	High

Reliability

AGENTREL01: How do I develop reliable agentic systems?

ID	Best practice	Risk
AGENTREL01-BP01	Implement a resilient messaging layer	High

ID	Best practice	Risk
AGENTREL01-BP02	Establish modular, fault-isolated layers	High
AGENTREL01-BP03	Design specialized agents following actor model principles	Medium
AGENTREL01-BP04	Standardize communication protocols	Medium
AGENTREL01-BP05	Implement adaptive provisioning	Medium

AGENTREL02: How do you develop agentic systems that reliably execute tasks with predictable outcomes?

ID	Best practice	Risk
AGENTREL02-BP01	Design agents for specific and atomic tasks	Medium
AGENTREL02-BP02	Limit agent permissions to minimum required access	High
AGENTREL02-BP03	Implement behavioral anomaly detection and monitoring	High
AGENTREL02-BP04	Develop clear instruction protocols for agents	Medium
AGENTREL02-BP05	Establish tiered human oversight and approval workflows	High

AGENTREL03: How do you support agent memory and state remaining reliably accessible throughout the agent lifecycle?

ID	Best practice	Risk
AGENTREL03-BP01	Design an information classification model to identify short-term and long-term memories	Medium
AGENTREL03-BP02	Architect fault-tolerant memory stores with redundancy and failover	High
AGENTREL03-BP03	Implement comprehensive state management and checkpoint-based recovery	High
AGENTREL03-BP04	Implement graceful degradation for memory and state operations	Medium

AGENTREL04: How do you orchestrate multi-agent systems to reliably execute tasks?

ID	Best practice	Risk
AGENTREL04-BP01	Implement the arbiter agent pattern for coordinated multi-agent systems	High
AGENTREL04-BP02	Classify agents with a comprehensive capability taxonomy	Medium

ID	Best practice	Risk
AGENTREL04-BP03	Implement fallback mechanisms and graceful degradation for collaborative workflows	High
AGENTREL04-BP04	Implement resilient control planes for agent coordination	High

AGENTREL05: How do you implement reliable agent cognition that accesses the right data at the right time?

ID	Best practice	Risk
AGENTREL05-BP01	Design modular, fault-tolerant agentic reasoning components	Medium
AGENTREL05-BP02	Facilitate reliable adaptation through evaluation-driven improvement cycles	Medium
AGENTREL05-BP03	Ground agent cognition in real information	High

AGENTREL06: How do agents integrate effectively with existing systems without impacting the reliability of established processes?

ID	Best practice	Risk
AGENTREL06-BP01	Develop agent-based integrations with existing or legacy systems	Medium

ID	Best practice	Risk
AGENTREL06-BP02	Establish fallback mechanisms for legacy system degradation	High
AGENTREL06-BP03	Regularly test degraded system performance	Medium
AGENTREL06-BP04	Implement idempotent task execution patterns	High
AGENTREL06-BP05	Implement dynamic capability toggling	Medium

AGENTREL07: How do fault tolerant agent systems recover?

ID	Best practice	Risk
AGENTREL07-BP01	Design workflows in stages with incremental recovery	High
AGENTREL07-BP02	Enable automatic recovery from agent execution failures	High
AGENTREL07-BP03	Implement distributed tracing to track system dependencies and facilitate recovery	High

AGENTREL08: How do agents determine when and where graceful degradation is appropriate?

ID	Best practice	Risk
AGENTREL08-BP01	Establish consistent configuration management practices	Medium

ID	Best practice	Risk
AGENTRELO8-BP02	Implement agent tracing for telemetry throughout agent processing	High
AGENTRELO8-BP03	Architect agent systems with resource isolation and contention mitigation	High
AGENTRELO8-BP04	Track agent memory utilization metrics	Medium

Performance efficiency

AGENTPERF01: How do you plan strategically for agent performance and establish measurement practices?

ID	Best practice	Risk
AGENTPERF01-BP01	Define performance-aligned success criteria for agent workloads	High
AGENTPERF01-BP02	Implement comprehensive performance telemetry	High
AGENTPERF01-BP03	Profile end-to-end agent latency and identify optimization targets	High

AGENTPERF02: How do you optimize core agent processing and cognitive pipelines?

ID	Best practice	Risk
AGENTPERF02-BP01	Design efficient reasoning pipelines	High
AGENTPERF02-BP02	Implement task-appropriate model selection strategies	High
AGENTPERF02-BP03	Optimize agent execution paths for reduced latency	High
AGENTPERF02-BP04	Optimize streaming responses and time-to-first-token for agent interactions	High

AGENTPERF03: How do you optimize memory management, context windows, and retrieval-augmented generation?

ID	Best practice	Risk
AGENTPERF03-BP01	Implement tiered memory management systems	High
AGENTPERF03-BP02	Optimize context window utilization and prompt management	High
AGENTPERF03-BP03	Optimize RAG retrieval pipelines for latency and precision	High

ID	Best practice	Risk
AGENTPERF03-BP04	Establish efficient agent caching and data access patterns	Medium
AGENTPERF03-BP05	Implement agentic retrieval patterns for dynamic, agent-driven knowledge access	High

AGENTPERF04: How do you achieve efficient communication and protocol usage across agent interactions?

ID	Best practice	Risk
AGENTPERF04-BP01	Optimize asynchronous message handling patterns	High
AGENTPERF04-BP02	Implement efficient protocol-based agent communications	Medium
AGENTPERF04-BP03	Design high-performing event-driven integration patterns	Medium

AGENTPERF05: How do you optimize workflow orchestration and multi-agent collaboration for performance?

ID	Best practice	Risk
AGENTPERF05-BP01	Design efficient workflow orchestration patterns	High
AGENTPERF05-BP02	Implement optimized multi-agent collaboration models	High

ID	Best practice	Risk
AGENTPERF05-BP03	Optimize multi-stage AI pipeline execution	Medium
AGENTPERF05-BP04	Implement efficient agent delegation and handoff patterns	Medium

AGENTPERF06: How do you optimize tool integrations and framework usage for agent performance?

ID	Best practice	Risk
AGENTPERF06-BP01	Design optimized tool integration strategies	Medium
AGENTPERF06-BP02	Implement efficient tool invocation patterns	Medium
AGENTPERF06-BP03	Optimize meta-tool utilization and tool chaining	Low

AGENTPERF07: How do you manage multitenant performance isolation and optimize resource utilization?

ID	Best practice	Risk
AGENTPERF07-BP01	Design efficient multitenant agent deployment models	High
AGENTPERF07-BP02	Implement tenant-aware performance isolation and throttling	High

Cost optimization

AGENTCOST01: How do you optimize agent reasoning and execution costs?

ID	Best practice	Risk
AGENTCOST01-BP01	Use the reflection pattern to design efficient agent reasoning loops	Medium
AGENTCOST01-BP02	Optimize multi-agent collaboration cost through efficient handoff patterns	High
AGENTCOST01-BP03	Implement cost-effective patterns like hybrid supervisor for multi-agent coordination	Medium
AGENTCOST01-BP04	Design agent hierarchies and delegation patterns that reduce coordination overhead	Medium

AGENTCOST02: How do you optimize agent model invocation and token consumption costs?

ID	Best practice	Risk
AGENTCOST02-BP01	Architect tiered model selection for cost-performance optimization	High
AGENTCOST02-BP02	Cost optimize token consumption through efficient prompt engineering	Medium

ID	Best practice	Risk
AGENTCOST02-BP03	Use intelligent caching to reduce redundant model invocations	High
AGENTCOST02-BP04	Implement model customization for long-term cost reduction	High

AGENTCOST03: How do you manage agent memory and state costs efficiently?

ID	Best practice	Risk
AGENTCOST03-BP01	Design cost-effective retrieval systems with tiered memory	Medium
AGENTCOST03-BP02	Cost optimize through intelligent compression and pruning of context windows	High
AGENTCOST03-BP03	Implement cost-optimized state persistence and lifecycle management	Medium

AGENTCOST04: How do you optimize agent tool invocation?

ID	Best practice	Risk
AGENTCOST04-BP01	Design cost effective tool selection to minimize unnecessary invocations	High

ID	Best practice	Risk
AGENTCOST04-BP02	Cost optimize tool serving through serverless and resource sharing	High
AGENTCOST04-BP03	Implement intelligent caching and failure handling for tool results	High

AGENTCOST05: How do you implement cost attribution?

ID	Best practice	Risk
AGENTCOST05-BP01	Establish agent-level reasoning cost tracking and attribution	Medium
AGENTCOST05-BP02	Implement distributed cost tracing for multi-agent workflows	High
AGENTCOST05-BP03	Design tenant-aware cost allocation for AaaS pricing models	High
AGENTCOST05-BP04	Create chargeback and ROI reporting	Medium

AGENTCOST06: How do you optimize agent discovery registry and deployment costs?

ID	Best practice	Risk
AGENTCOST06-BP01	Implement lightweight discovery and registry for cost-effective collaboration	Medium
AGENTCOST06-BP02	Cost optimize versioning and deployment through efficient artifact management	Medium
AGENTCOST06-BP03	Design cost-efficient initialization through warm pools and caching	High

AGENTCOST07: How do you establish agent cost governance and continuous optimization?

ID	Best practice	Risk
AGENTCOST07-BP01	Implement automated cost controls with intelligent cutoffs	High
AGENTCOST07-BP02	Establish proactive anomaly detection for agent cost patterns	High
AGENTCOST07-BP03	Create systematic optimization feedback loops for continuous improvement	High

Sustainability

AGENTSUS01: How do you build sustainable and repeatable frameworks for managing compute, memory, and other shareable agent resources?

ID	Best practice	Risk
AGENTSUS01-BP01	Design specialized agents with explicit resource boundaries	High
AGENTSUS01-BP02	Implement reusable workflow patterns	Medium
AGENTSUS01-BP03	Optimize resource utilization through shared services	Medium
AGENTSUS01-BP04	Scale cognitive processing pathways appropriately	High
AGENTSUS01-BP05	Adopt specification-driven tasks for frontier agents and long-running workflows	High

AGENTSUS02: How do I establish sustainable frameworks for agent dependencies?

ID	Best practice	Risk
AGENTSUS02-BP01	Optimize context management and memory utilization	Medium
AGENTSUS02-BP02	Establish efficient agent caching strategies	Medium

ID	Best practice	Risk
AGENTSUS02-BP03	Appropriately scale data, networking, and compute dependencies	Medium
AGENTSUS02-BP04	Measure and optimize the environmental footprint of agent workloads	Medium

AGENTSUS03: How do I establish durable patterns for agent interactions with users and business processes?

ID	Best practice	Risk
AGENTSUS03-BP01	Maintain organizational skills and competencies	High
AGENTSUS03-BP02	Build agents to mirror your organizational skills and competencies	Medium
AGENTSUS03-BP03	Maintain comprehensive specifications for agents and agentic systems	High
AGENTSUS03-BP04	Decommission unused agents and prevent agent sprawl	Medium

Summary

Best practice summary by pillar

Pillar	Questions	Best practices	High Risk	Medium Risk	Low Risk
Operational Excellence	7	26	18	8	0
Security	9	30	19	11	0
Reliability	8	33	18	15	0
Performance Efficiency	7	24	16	7	1
Cost Optimization	7	24	14	10	0
Sustainability	3	13	5	8	0
Total	41	150	90	59	1

Document revisions

The following table describes the documentation releases for the Agentic AI Lens.

Change	Description	Date
Initial release	Initial release of the Agentic AI Lens.	June 10, 2026

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided "as is" without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

© 2026 Amazon Web Services, Inc. or its affiliates. All rights reserved.

AWS Glossary

For the latest AWS terminology, see the [AWS glossary](#) in the *AWS Glossary Reference*.