Choosing a Git branching strategy for multi-account DevOps environments

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Choosing a Git branching strategy for multi-account DevOps environments

Table of Contents

Choosing a Git branching strategy for multi-account DevOps environments

Amazon Web Services (contributors)

February 2024 (document history)

Moving to a cloud-based approach and delivering software solutions on AWS can be transformative. It might require changes to your software development lifecycle process. Typically, multiple AWS accounts are used during the development process in the AWS Cloud. Choosing a compatible Git branching strategy to pair with your DevOps processes is essential to success. Choosing the right Git branching strategy for your organization helps you concisely communicate DevOps standards and best practices across development teams. Git branching can be simple in a single environment, but it can become confusing when applied across multiple environments, such as sandbox, development, testing, staging, and production environments. Having multiple environments increases the complexity of the DevOps implementation.

This guide provides visual diagrams of Git branching strategies that show how an organization can implement a multi-account DevOps process. Visual guides help teams understand how to merge their Git branching strategies with their DevOps practices. Using a standard branching model, like Gitflow, GitHub Flow, or Trunk, for managing the source code repository helps development teams align their work. These teams can also use standard Git training resources on the internet to understand and implement those models and strategies.

For DevOps best practices on AWS, review the DevOps Guidance in AWS Well-Architected. As you review this guide, use due diligence to select the right branching strategy for your organization. Some strategies might fit your use case better than others.

Objectives

This guide is part of a documentation series about choosing and implementing DevOps branching strategies for organizations with multiple AWS accounts. This series is designed to help you apply the strategy that best meets your requirements, goals, and best practices from the outset, to streamline your experience in the AWS Cloud. This guide does not contain DevOps executable scripts because they vary based on the continuous integration and continuous delivery (CI/CD) engine and technology frameworks that your organization uses.

This guide explains the differences between three common Git branching strategies: GitHub Flow, Gitflow, and Trunk. The recommendations in this guide help teams identify a branching strategy that aligns with their organizational goals. After reviewing this guide, you should be able to choose a branching strategy for your organization. After choosing a strategy, you can use one of the following patterns to help you implement that strategy with your development teams:

- Implement a Trunk branching strategy for multi-account DevOps environments
- Implement a GitHub Flow branching strategy for multi-account DevOps environments
- Implement a Gitflow branching strategy for multi-account DevOps environments

It's important to note that what works for one organization, team, or project might not be suitable for others. The choice between Git branching strategies depends on various factors, such as team size, project requirements, and the desired balance between collaboration, integration frequency, and release management.

Using CI/CD practices

AWS recommends that you implement continuous integration and continuous delivery (CI/CD), which is the process of automating the software release lifecycle. It automates much or all of the manual DevOps processes that are traditionally required to get new code from development into production. A CI/CD pipeline encompasses the sandbox, development, testing, staging, and production environments. In each environment, the CI/CD pipeline provisions any infrastructure that is needed to deploy or test the code. By using CI/CD, development teams can make changes to code that are then automatically tested and deployed. CI/CD pipelines also provide governance and guardrails for development teams. They enforce consistency, standards, best practices, and minimum acceptance levels for feature acceptance and deployment. For more information, see Practicing Continuous Integration and Continuous Delivery on AWS.

All of the branching strategies discussed in this guide are well suited to CI/CD practices. The complexity of the CI/CD pipeline increases with the complexity of the branching strategy. For example, Gitflow is the most complex branching strategy discussed in this guide. CI/CD pipelines for this strategy require more steps (such as for compliance reasons), and they must support multiple, simultaneous production releases. Using CI/CD also becomes more important as the complexity of the branching strategy increases. This is because CI/CD establishes guardrails and mechanisms for development teams that prevents developers from intentionally or unintentionally circumnavigating the defined process.

AWS offers a suite of developer services that are designed to help you build CI/CD pipelines. For example, AWS CodePipeline is a fully managed continuous delivery service that helps you automate your release pipelines for fast and reliable application and infrastructure updates. AWS CodeBuild compiles source code, runs tests, and produces ready-to-deploy software packages. For more information, see Developer Tools on AWS.

Understanding the DevOps environments

To understand the branching strategies, you must understand the purpose and activities that occur in each environment. Establishing several environments helps you separate the development activities into stages, monitor those activities, and prevent the unintentional release of unapproved features. You can have one or more AWS accounts in each environment.

Most organizations have several environments outlined for use. However, the number of environments can vary by organization and according to software development policies. This documentation series assumes that you have the following five, common environments that span your development pipeline, although they might be called by different names:

- **Sandbox** – An environment where developers write code, make mistakes, and perform proof of concept work.
- **Development** – An environment where developers integrate their code to confirm that it all works as a single, cohesive application.
- **Testing** – An environment where QA teams or acceptance testing takes place. Teams often do performance or integration testing in this environment.
- **Staging** – A preproduction environment where you validate that the code and infrastructure perform as expected under production-equivalent circumstances. This environment is configured to be as similar as possible to the production environment.
- **Production** – An environment that handles traffic from your end users and customers.

This section describes each environment in detail. It also describes the build steps, deployment steps, and exit criteria for each environment so that you can proceed to the next. The following image shows these environments in sequence.

Topics in this section:

- Sandbox environment
- Development environment

- Testing environment
- Staging environment
- Production environment

Sandbox environment

The *sandbox environment* is where developers write code, make mistakes, and perform proof of concept work. You can deploy to a sandbox environment from a local workstation or through a script on a local workstation.

Access

Developers should have full access to the sandbox environment.

Build steps

Developers manually run the build on their local workstations when they are ready to deploy changes to the sandbox environment.

1. Use git-secrets (GitHub) to scan for sensitive information
2. Lint the source code
3. Build and compile the source code, if applicable
4. Perform unit testing
5. Perform code coverage analysis
6. Perform static code analysis
7. Build infrastructure as code (IaC)
8. Perform IaC security analysis
9. Extract open source licenses
10. Publish build artifacts

Deployment steps

If you're using the Gitflow or Trunk models, the deployment steps automatically initiate when a `feature` branch is successfully built in the sandbox environment. If you're using the GitHub

Flow model, then you manually perform the following deployment steps. The following are the deployment steps in the sandbox environment:

1. Download published artifacts
2. Perform database versioning
3. Perform IaC deployment
4. Perform integration testing

Expectations before moving to the development environment

- Successful build of the `feature` branch in the sandbox environment
- A developer has manually deployed and tested the feature in the sandbox environment

Development environment

The *development environment* is where developers integrate their code together to ensure it all works as one cohesive application. In Gitflow, the development environment contains the latest features included by merge request and are ready for release. In GitHub Flow and Trunk strategies, the development environment is considered to be a testing environment, and the code base might be unstable and unsuitable for deployment to production.

Access

Assign permissions according to the principle of least privilege. *Least privilege* is the security best practice of granting the minimum permissions required to perform a task. Developers should have less access to the development environment than they have to the sandbox environment.

Build steps

Creating a merge request to the `develop` branch (Gitflow) or the `main` branch (Trunk or GitHub Flow) automatically starts the build.

1. Use git-secrets (GitHub) to scan for sensitive information
2. Lint the source code
3. Build and compile the source code, if applicable
4. Perform unit testing

5. Perform code coverage analysis
6. Perform static code analysis
7. Build IaC
8. Perform IaC security analysis
9. Extract open source licenses

Deployment steps

If you're using the Gitflow model, the deployment steps automatically initiate when a `develop` branch is successfully built in the development environment. If you're using the GitHub Flow model or Trunk model, then the deployment steps automatically initiate when a merge request is created against the `main` branch. The following are the deployment steps in the development environment:

1. Download the published artifacts from the build steps
2. Perform database versioning
3. Perform IaC deployment
4. Perform integration tests

Expectations before moving to the testing environment

- Successful build and deployment of the `develop` branch (Gitflow) or the `main` branch (Trunk or GitHub Flow) in the development environment
- Unit testing passes at 100%
- Successful IaC build
- Deployment artifacts were successfully created
- A developer has performed a manual verification to confirm that the feature is functioning as expected

Testing environment

Quality assurance (QA) personnel use the testing environment to validate features. They approve the changes after they finish testing. When they approve, the branch moves on to the next

environment, staging. In Gitflow, this environment and others above it are only available for deployment from `release` branches. A `release` branch is based on a `develop` branch that contains the planned features.

Access

Assign permissions according to the principle of least privilege. Developers should have less access to the testing environment than they have to the development environment. QA personnel require sufficient permissions to test the feature.

Build steps

The build process in this environment is only applicable for bugfixes when using the Gitflow strategy. Creating a merge request to the `bugfix` branch automatically starts the build.

1. Use git-secrets (GitHub) to scan for sensitive information
2. Lint the source code
3. Build and compile the source code, if applicable
4. Perform unit testing
5. Perform code coverage analysis
6. Perform static code analysis
7. Build IaC
8. Perform IaC security analysis
9. Extract open source licenses

Deployment steps

Automatically initiate deployment of the `release` branch (Gitflow) or the `main` branch (Trunk or GitHub Flow) in the testing environment after deployment in the development environment. The following are the deployment steps in the testing environment:

1. Deploy the `release` branch (Gitflow) or `main` branch (Trunk or GitHub Flow) in the testing environment
2. Pause for manual approval by designated personnel

3. Download published artifacts
4. Perform database versioning
5. Perform IaC deployment
6. Perform integration tests
7. Perform performance tests
8. Quality assurance approval

Expectations before moving to the staging environment

- The development and QA teams have performed sufficient testing to satisfy your organization's requirements.
- The development team has resolved any discovered bugs through a `bugfix` branch.

Staging environment

The *staging environment* is configured to be the same as the production environment. For example, the data setup should be similar in scope and size to production workloads. Use the staging environment to verify that code and infrastructure operate as expected. This environment is also the preferred choice for business use cases, such as previews or customer demonstrations.

Access

Assign permissions according to the principle of least privilege. Developers should have the same access to the staging environment as they do the production environment.

Build steps

None. The same artifacts that were used in the testing environment are reused in the staging environment.

Deployment steps

Automatically initiate deployment of the `release` branch (Gitflow) or the `main` branch (Trunk or GitHub Flow) in the staging environment after approval and deployment in the testing environment. The following are the deployment steps in the staging environment:

1. Deploy the `release` branch (Gitflow) or `main` branch (Trunk or GitHub Flow) in the staging environment
2. Pause for manual approval by designated personnel
3. Download published artifacts
4. Perform database versioning
5. Perform IaC deployment
6. (Optional) Perform integration testing
7. (Optional) Perform load testing
8. Obtain approval from the required development, QA, product, or business approvers

Expectations before moving to the production environment

- A production-equivalent release has been deployed successfully to the staging environment
- (Optional) Integration and load testing were successful

Production environment

The *production environment* supports the released product, handling real data by real clients. This is a protected environment that is assigned access by least privilege and elevated access should only be allowed through an audited exception process for a limited period of time.

Access

In the production environment, developers should have limited, read-only access in the AWS Management Console. For example, developers should be able to access log data for day-to-day operations. All releases to production should be gated by an approval step prior to deployment.

Build steps

None. The same artifacts that were used in the testing and staging environments are reused in the production environment.

Deployment steps

Automatically initiate deployment of the `release` branch (Gitflow) or the `main` branch (Trunk or GitHub Flow) in the production environment after approval and deployment in the staging environment. The following are the deployment steps in the production environment:

1. Deploy the `release` branch (Gitflow) or `main` branch (Trunk or GitHub Flow) in the production environment
2. Pause for manual approval by designated personnel
3. Download published artifacts
4. Perform database versioning
5. Perform IaC deployment

Best practices for Git-based development

To successfully adopt Git-based development, it's important to follow a set of best practices that promote collaboration, maintain code quality, and support continuous integration and continuous delivery (CI/CD). In addition to the best practices in this guide, review the AWS Well-Architected DevOps Guidance. The following are some key best practices for Git-based development on AWS:

- **Keep changes small and frequent** – Encourage developers to commit small, incremental changes or features. This reduces the risk of merge conflicts and makes it easier to identify and fix issues quickly.
- **Use feature toggles** – To manage the release of incomplete or experimental features, use feature toggles or feature flags. This helps you hide, enable, or disable specific features in production without affecting the main branch's stability.
- **Maintain a robust test suite** – A comprehensive, well-maintained test suite is crucial for detecting issues early and verifying that the code base remains stable. Invest in test automation and prioritize fixing any failing tests.
- **Embrace continuous integration** – Use continuous integration tools and practices to automatically build, test, and integrate code changes into the `develop` branch (Gitflow) or `main` branch (Trunk or GitHub Flow). This helps you catch issues early and streamlines the development process.
- **Perform code reviews** – Encourage peer reviews of code to maintain quality, share knowledge, and catch potential issues before they're integrated into the `main` branch. Use pull requests or other code review tools to facilitate this process.
- **Monitor and fix broken builds** – When a build breaks or tests fail, prioritize fixing the issue as soon as possible. This keeps the `develop` branch (Gitflow) or `main` branch (Trunk or GitHub Flow) in a releasable state and minimizes the impact on other developers.
- **Communicate and collaborate** – Promote open communication and collaboration among team members. Make sure that developers are aware of ongoing work and changes being made to the code base.
- **Refactor continuously** – Regularly refactor the code base to improve its maintainability and reduce technical debt. Encourage developers to leave the code in a better state than they found it.
- **Use short-lived branches for complex tasks** – For larger or more complex tasks, use short-lived branches (also known as *task branches*) to work on the changes. However, make sure to keep the

branch lifespan short, typically less than a day. Merge the changes back into the `develop` branch (Gitflow) or `main` branch (Trunk or GitHub Flow) as soon as possible. Smaller and more frequent merges and reviews are easier for a team to consume and process than one large merge request.

- **Train and support the team** – Provide training and support to developers who are new to Git-based development or who require guidance in adopting its best practices.

Git branching strategies

In order of least to most complex, this guide describes the following Git-based branching strategies in detail:

- **Trunk** – Trunk-based development is a software development practice in which all developers work on a single branch, typically called the `trunk` or `main` branch. The idea behind this approach is to keep the code base in a continuously releasable state by integrating code changes frequently and relying on automated testing and continuous integration.
- **GitHub Flow** – GitHub Flow is a lightweight, branch-based workflow that was developed by GitHub. It is based on the idea of short-lived `feature` branches. When a feature is complete and ready to be deployed, the feature is merged into the `main` branch.
- **Gitflow** – With a Gitflow approach, development is completed in individual feature branches. After approval, you merge `feature` branches into an integration branch that is usually named `develop`. When enough features have accumulated in the `develop` branch, a `release` branch is created to deploy the features to upper environments.

Each branching strategy has advantages and disadvantages. Although they all use the same environments, they don't all use the same branches or manual approval steps. In this section of the guide, review each branching strategy in detail so that you're familiar with its nuances and can evaluate whether it fits your organization's use case.

Topics in this section:

Trunk branching strategy

Trunk-based development is a software development practice in which all developers work on a single branch, typically called the `trunk` or `main` branch. The idea behind this approach is to keep the code base in a continuously releasable state by integrating code changes frequently and relying on automated testing and continuous integration.

In trunk-based development, developers commit their changes to the `main` branch multiple times a day, aiming for small, incremental updates. This enables quick feedback loops, reduces the risk of merge conflicts, and fosters collaboration among team members. The practice emphasizes the importance of a well-maintained test suite because it relies on automated testing to catch potential issues early and make sure that the code base remains stable and releasable.

Trunk-based development is often contrasted with *feature-based development* (also known as *feature branching* or *feature-driven development*), where each new feature or bug fix is developed in its own dedicated branch, separate from the main branch. The choice between trunk-based development and feature-based development depends on factors such as team size, project requirements, and the desired balance between collaboration, integration frequency, and release management.

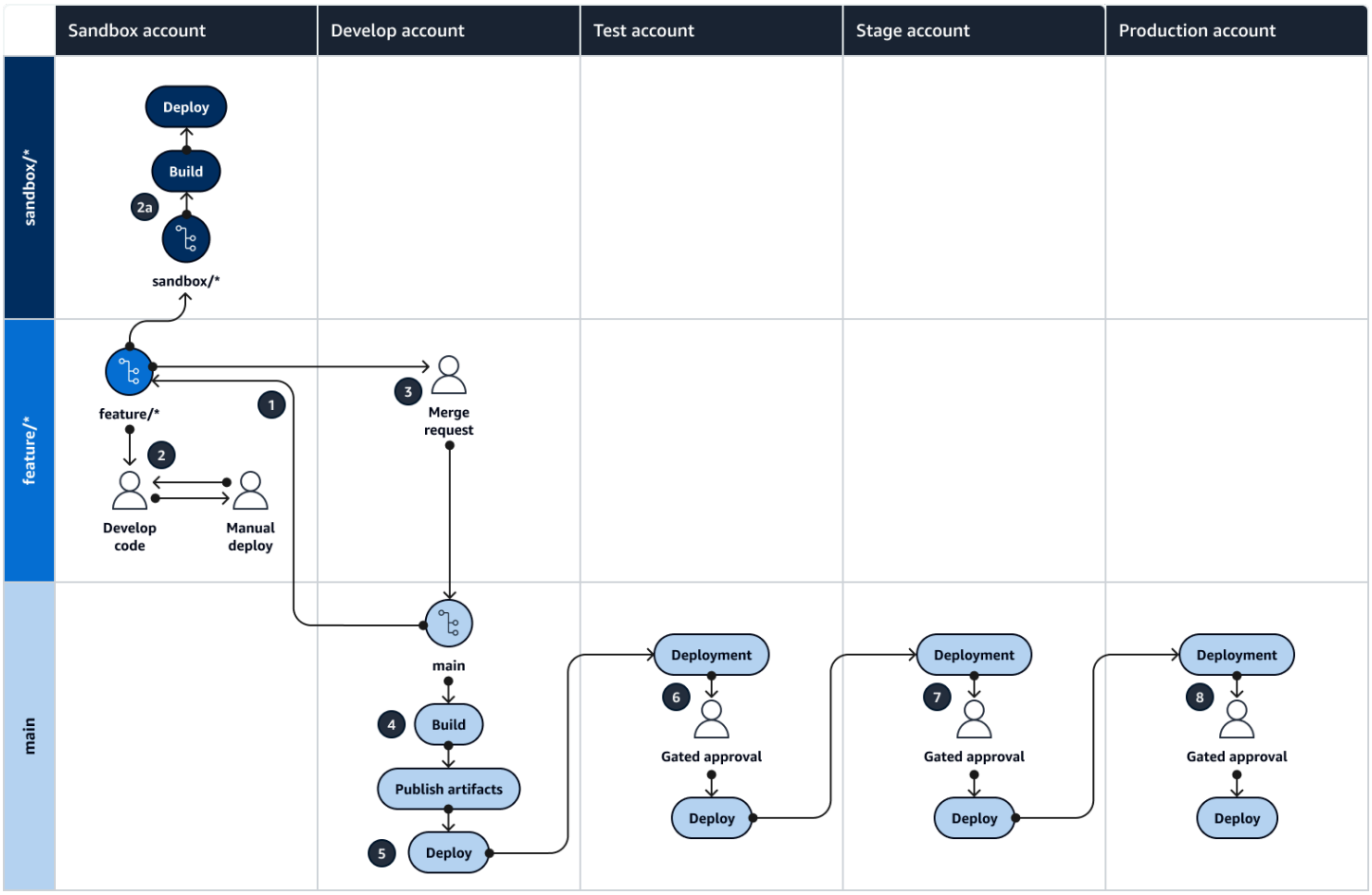
For more information about the Trunk branching strategy, see the following resources:

- Implement a Trunk branching strategy for multi-account DevOps environments (AWS Prescriptive Guidance)
- Introduction to Trunk-Based Development (Trunk Based Development website)

Topics in this section:

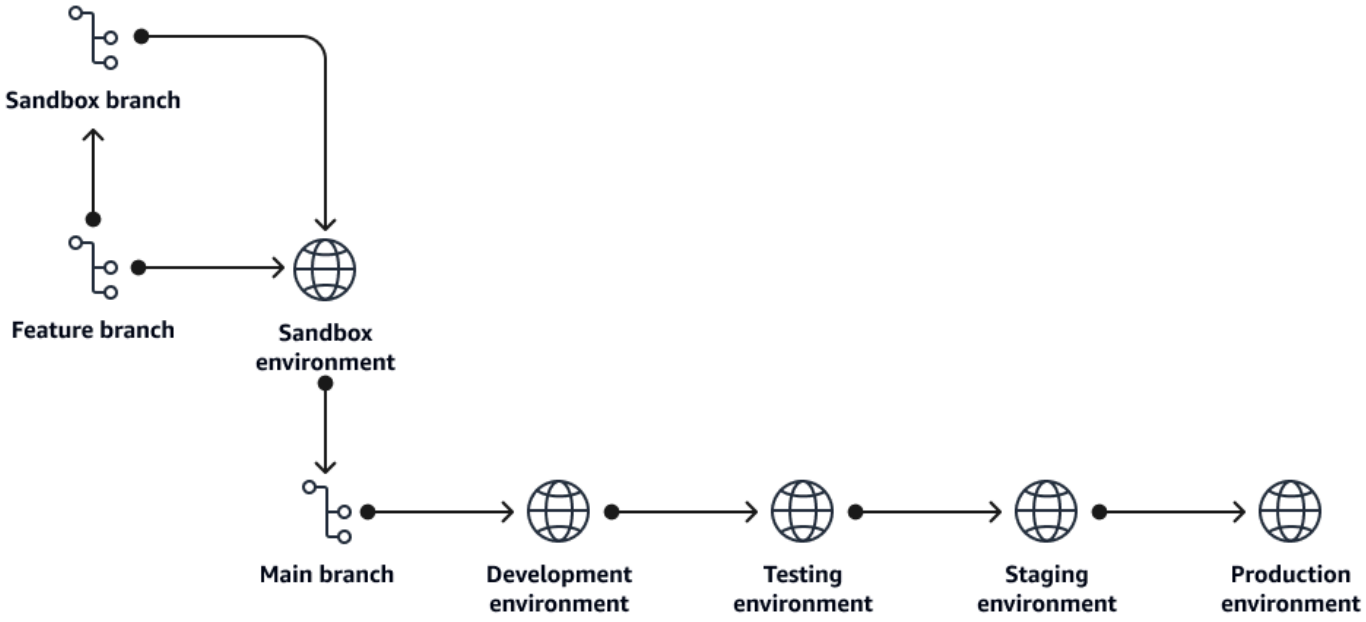
Visual overview of the Trunk strategy

The following diagram can be used like a Punnett square (Wikipedia) to understand the Trunk branching strategy. Line up the branches on the vertical axis with the AWS environments on the horizontal axis to determine what actions to perform in each scenario. The circled numbers guide you through the sequence of actions represented in the diagram. This diagram shows the development workflow of a Trunk branching strategy, from a `feature` branch in the sandbox environment to production release of the `main` branch. For more information about the activities that occur in each environment, see DevOps environments in this guide.



Branches in a Trunk strategy

A Trunk branching strategy commonly has the following branches.



feature branch

You develop features or create a hotfix in a `feature` branch. To create a `feature` branch, you branch off of the `main` branch. Developers iterate, commit, and test code in a `feature` branch. When a feature is complete, the developer promotes the feature. There are only two paths forward from a `feature` branch:

- Merge into the `sandbox` branch
- Create a merge request into the `main` branch

Naming convention:	`feature/<story number>_<developer initials>_<descriptor>`
Naming convention example:	`feature/123456_MS_Implement_Feature_A`

sandbox branch

This branch is a non-standard trunk branch, but it is useful for CI/CD pipeline development. The `sandbox` branch is primarily used for the following purposes:

- Perform a full deployment to the sandbox environment by using the CI/CD pipelines
- Develop and test a pipeline before submitting merge requests for full testing in a lower environment, such as development or testing.

Sandbox branches are temporary in nature and are intended to be short-lived. They should be deleted after the specific testing is complete.

Naming convention:	`sandbox/<story number>_<developer initials>_<descriptor>`
Naming convention example:	`sandbox/123456_MS_Test_Pipeline_Deploy`

main branch

The `main` branch always represents the code that is running in production. Code is branched from `main`, developed, and then merged back to `main`. Deployments from `main` could target any environment. To protect against deletion, enable branch protection for the `main` branch.

Naming convention:	`main`

hotfix branch

There is no dedicated `hotfix` branch in a trunk-based workflow. Hotfixes use `feature` branches.

Advantages and disadvantages of the Trunk strategy

The Trunk branching strategy is well suited for smaller, mature, development teams that have strong communication skills. It also works well if you have continuous, rolling feature releases for the application. It is not well suited if you have large or fragmented development teams or if you have expansive, scheduled feature releases. Merge conflicts will occur in this model, so be aware that resolution of merge conflicts is a key skill. All team members must be trained accordingly.

Advantages

Trunk-based development offers several advantages that can improve the development process, streamline collaboration, and enhance the overall quality of the software. The following are some of the key benefits:

- **Faster feedback loops** – With trunk-based development, developers integrate their code changes frequently, often multiple times a day. This enables faster feedback regarding potential issues and helps developers identify and fix problems more quickly than they would in a feature-based development model.
- **Reduced merge conflicts** – In trunk-based development, the risk of large, complicated merge conflicts is minimized because changes are integrated continuously. This helps maintain a cleaner code base and reduces the amount of time spent resolving conflicts. Resolving conflicts can be both time-consuming and error-prone in feature-based development.
- **Improved collaboration** – Trunk-based development encourages developers to work together on the same branch, promoting better communication and collaboration within the team. This can lead to faster problem-solving and a more cohesive team dynamic.
- **Easier code reviews** – Because code changes are smaller and more frequent in trunk-based development, it can be easier to conduct thorough code reviews. Smaller changes are generally easier to understand and review, leading to more effective identification of potential issues and improvements.
- **Continuous integration and delivery** – Trunk-based development supports the principles of continuous integration and continuous delivery (CI/CD). By keeping the code base in a releasable state and integrating changes frequently, teams can more easily adopt CI/CD practices, which leads to faster deployment cycles and improved software quality.
- **Enhanced code quality** – With frequent integration, testing, and code reviews, trunk-based development can contribute to better overall code quality. Developers can catch and fix issues more quickly, reducing the likelihood of technical debt accumulating over time.
- **Simplified branching strategy** – Trunk-based development simplifies the branching strategy by reducing the number of long-lived branches. This can make it easier to manage and maintain the code base, especially for large projects or teams.

Disadvantages

Trunk-based development does have some disadvantages, which can impact the development process and the team dynamics. The following are a few notable drawbacks:

- **Limited isolation** – Because all developers work on the same branch, their changes are immediately visible to everyone on the team. This can lead to interference or conflicts, causing unintended side effects or breaking the build. In contrast, feature-based development isolates changes better so that developers can work more independently.
- **Increased pressure on testing** – Trunk-based development relies on continuous integration and automated testing to catch issues quickly. However, this approach can put a lot of pressure on the testing infrastructure and requires a well-maintained test suite. If the tests aren't comprehensive or reliable, it can lead to undetected issues in the main branch.
- **Less control over releases** – Trunk-based development aims to keep the code base in a continuously releasable state. While this can be advantageous, it might not always be suitable for projects with strict release schedules or those that require specific features to be released together. Feature-based development provides more control over when and how features are released.
- **Code churn** – With developers constantly integrating changes into the main branch, trunk-based development can lead to increased code churn. This can make it difficult for developers to keep track of the current state of the code base and might cause confusion when trying to understand the effect of recent changes.
- **Requires a strong team culture** – Trunk-based development demands a high level of discipline, communication, and collaboration among team members. This can be challenging to maintain, particularly in larger teams or when working with developers who are less experienced with this approach.
- **Scalability challenges** – As the size of the development team grows, the number of code changes being integrated into the main branch can increase rapidly. This can lead to more frequent build breaks and test failures, making it difficult to keep the code base in a releasable state.

GitHub Flow branching strategy

GitHub Flow is a lightweight, branch-based workflow was developed by GitHub. GitHub Flow is based on the idea of short-lived feature branches that are merged into the main branch when the feature is complete and ready to be deployed. The key principles of GitHub Flow are:

- **Branching is lightweight** – Developers can create feature branches for their work with just a few clicks, improving the ability to collaborate and experiment without affecting the main branch.
- **Continuous deployment** – Changes are deployed as soon as they are merged into the main branch, which allows for rapid feedback and iteration.
- **Merge requests** – Developers use merge requests to initiate a discussion and review process before merging their changes into the main branch.

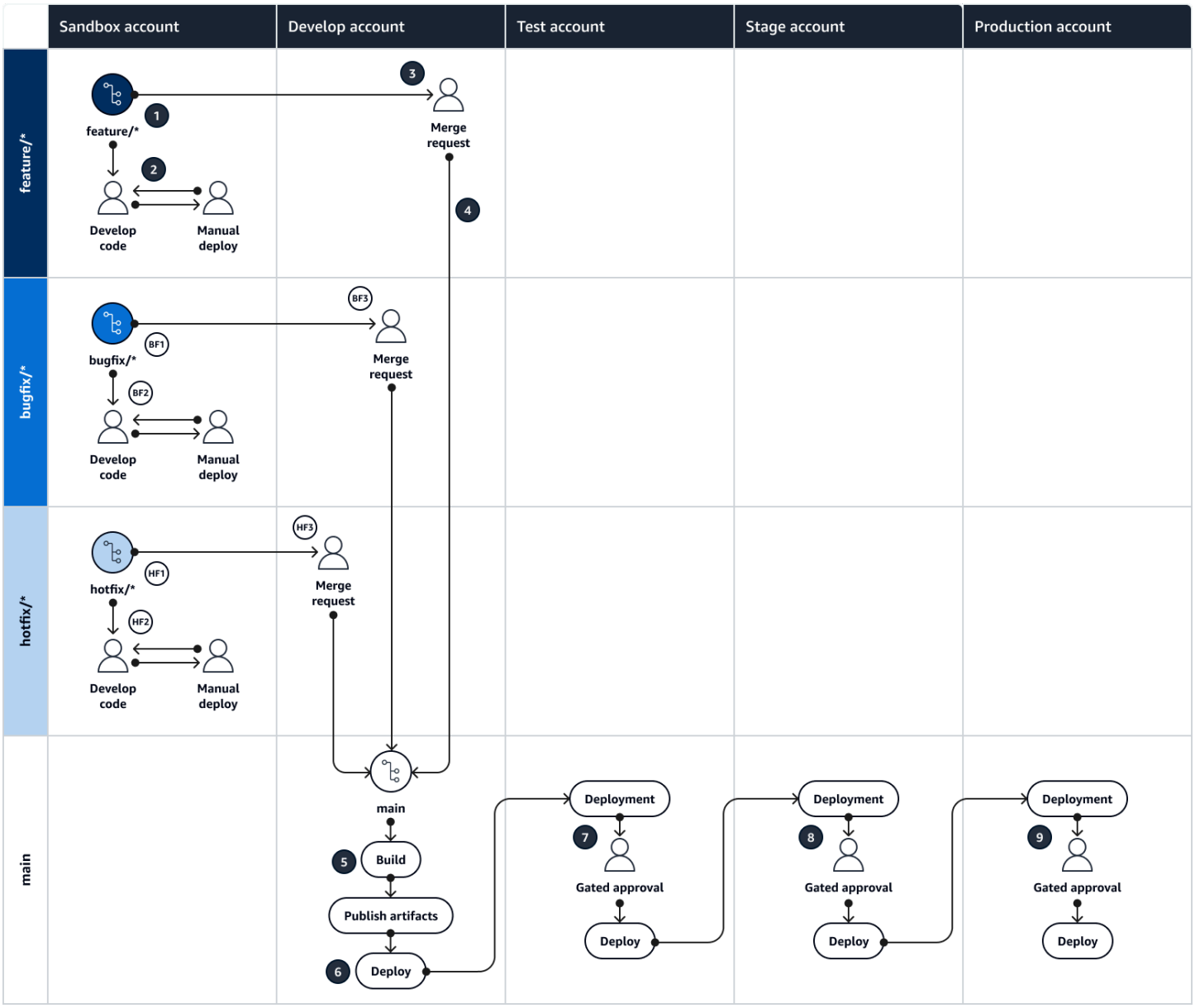
For more information about GitHub Flow, see the following resources:

- Implement a GitHub Flow branching strategy for multi-account DevOps environments (AWS Prescriptive Guidance)
- GitHub Flow Quickstart (GitHub documentation)
- Why GitHub Flow? (GitHub Flow website)

Topics in this section:

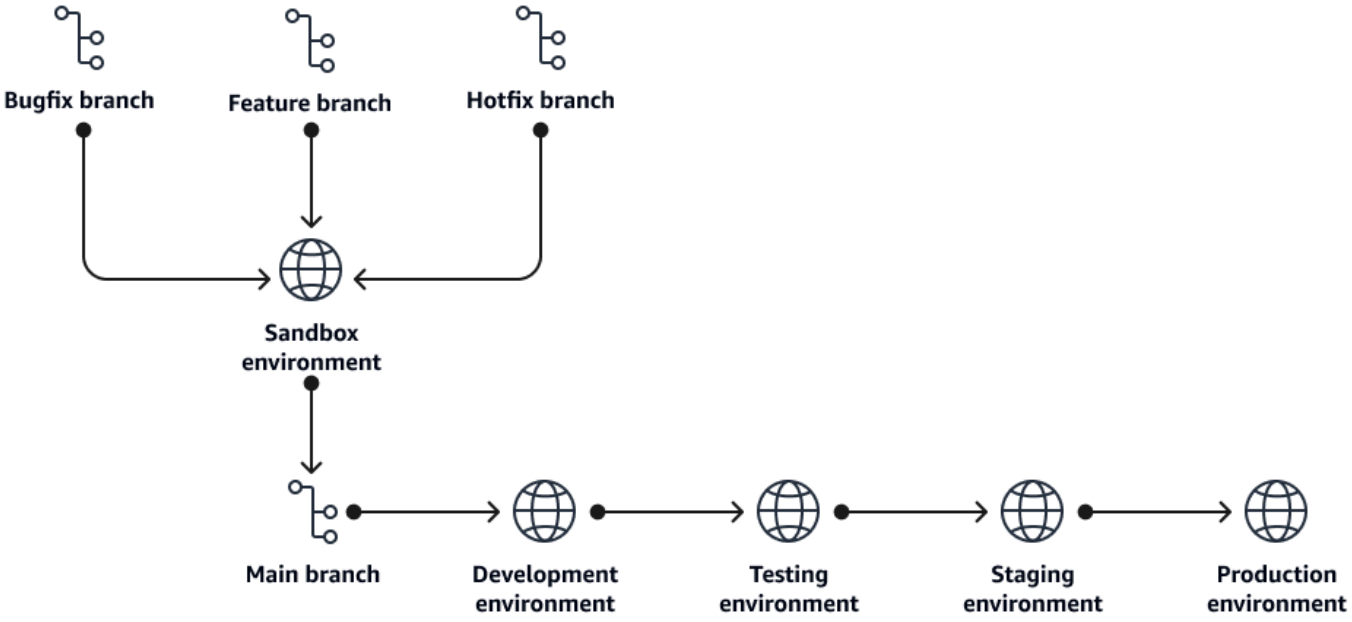
Visual overview of the GitHub Flow strategy

The following diagram can be used like a Punnett square to understand the GitHub Flow branching strategy. Line up the branches on the vertical axis with the AWS environments on the horizontal axis to determine what actions to perform in each scenario. The circled numbers guide you through the sequence of actions represented in the diagram. This diagram shows the development workflow of a GitHub Flow branching strategy, from a feature branch in the sandbox environment to production release of the main branch. For more information about the activities that occur in each environment, see DevOps environments in this guide.



Branches in a GitHub Flow strategy

A GitHub Flow branching strategy commonly has the following branches.



feature branch

You develop features in `feature` branches. To create a `feature` branch, you branch off of the `main` branch. Developers iterate, commit, and test the code in the `feature` branch. When a feature is complete, the developer promotes the feature by creating a merge request to `main`.

Naming convention:	`feature/<story number>_<developer initials>_<descriptor>`
Naming convention example:	`feature/123456_MS_Implement_Feature_A`

bugfix branch

The `bugfix` branch is used to fix issues. These branches are branched off of the `main` branch. After the bugfix is tested in sandbox or any of the lower environments, it can be promoted to higher environments by merging it to `main` through a merge request. This is a suggested naming convention for organization and tracking, this process could also be managed using a feature branch.

Naming convention:	`bugfix/<ticket number>_<developer initials>_<descriptor>`
Naming convention example:	`bugfix/123456_MS_Fix_Problem_A`

hotfix branch

The `hotfix` branch is used to resolve high impact critical issues with minimal delay between the development staff and the code deployed in production. These branches are branched off of the `main` branch. After the hotfix is tested in sandbox or any of the lower environments, it can be promoted to higher environments by merging it to `main` through a merge request. This is a suggested naming convention for organization and tracking, this process could also be managed using a feature branch.

Naming convention:	`hotfix/<ticket number>_<developer initials>_<descriptor>`
Naming convention example:	`hotfix/123456_MS_Fix_Problem_A`

main branch

The `main` branch always represents the code that is running in production. Code is merged into the `main` branch from `feature` branches by using merge requests. To protect against deletion and to prevent developers from pushing code directly to `main`, enable branch protection for the `main` branch.

Naming convention:	`main`

Advantages and disadvantages of the GitHub Flow strategy

The Github Flow branching strategy is well suited for smaller, mature, development teams that have strong communication skills. This strategy is well suited to teams that want to implement continuous delivery, and it is well supported by common CI/CD engines. GitHub Flow is

lightweight, doesn't have too many rules, and is capable of supporting fast-moving teams. It is not well suited if your teams have strict compliance or release processes to follow. Merge conflicts are common in this model and will likely happen often. Resolution of merge conflicts is a key skill, and you must train all team members accordingly.

Advantages

GitHub Flow offers several advantages that can improve the development process, streamline collaboration, and enhance the overall quality of the software. The following are some of the key benefits:

- **Flexible and lightweight** – GitHub Flow is a lightweight and flexible workflow that helps developers collaborate on software development projects. It allows for quick iteration and experimentation with minimal complexity.
- **Simplified collaboration** – GitHub Flow provides a clear and streamlined process for managing feature development. It encourages small, focused changes that can be quickly reviewed and merged, improving efficiency.
- **Clear version control** – With GitHub Flow, every change is made in a separate branch. This establishes a clear and traceable version control history. This helps developers track and understand changes, revert if necessary, and maintain a reliable code base.
- **Seamless continuous integration** – GitHub Flow integrates with continuous integration tools. Creation of pull requests can initiate automated testing and deployment processes. CI tools help you thoroughly test changes before they are merged into the `main` branch, reducing the risk of introducing bugs into the code base.
- **Rapid feedback and continuous improvement** – GitHub Flow encourages a rapid feedback loop by promoting frequent code reviews and discussions through pull requests. This facilitates early detection of issues, promotes knowledge sharing among team members, and ultimately leads to higher code quality and better collaboration within the development team.
- **Simplified rollbacks and reverts** – In the event that a code change introduces an unexpected bug or issue, GitHub Flow simplifies the process of rolling back or reverting the change. By having a clear history of commits and branches, it is easier to identify and revert problematic changes, helping to maintain a stable and functional code base.
- **Lightweight learning curve** – GitHub Flow can be easier to learn and adopt than Gitflow, especially for teams already familiar with Git and version control concepts. Its simplicity and intuitive branching model make it accessible for developers of varying experience levels, reducing the learning curve associated with adopting new development workflows.

- **Continuous development** – GitHub Flow empowers teams to embrace a continuous deployment approach by enabling the immediate deployment of every change as soon as it is merged into the `main` branch. This streamlined process eliminates unnecessary delays and makes sure that the latest updates and improvements are quickly made available to users. This results in a more agile and responsive development cycle.

Disadvantages

While GitHub Flow offers several advantages, it is important to consider its potential disadvantages as well:

- **Limited suitability for large projects** – GitHub Flow may not be as suitable for large-scale projects with complex code bases and multiple long-term feature branches. In such cases, a more structured workflow, like Gitflow, might provide better control over concurrent development and release management.
- **Lack of formal release structure** – GitHub Flow does not explicitly define a release process or support features such as versioning, hotfixes, or maintenance branches. This can be a limitation for projects that require strict release management or have a need for long-term support and maintenance.
- **Limited support for long-term release planning** – GitHub Flow focuses on short-lived feature branches, which might not align well with projects that require long-term release planning, such as those with strict roadmaps or extensive feature dependencies. Managing complex release schedules can be challenging within the constraints of GitHub Flow.
- **Potential for frequent merge conflicts** – Because GitHub Flow encourages frequent branching and merging, there is a possibility of encountering merge conflicts, especially in projects with a lot of development activity. Resolving these conflicts can be time-consuming and might require additional effort from the development team.
- **Lack of formalized workflow phases** – GitHub Flow does not define explicit phases for development, such as alpha, beta, or release candidate stages. This can make it harder to communicate the project's current state or the stability level of different branches or releases.
- **Impact of breaking changes** – Because GitHub Flow encourages merging changes into the `main` branch frequently, there is a higher risk of introducing breaking changes that affect the stability of the code base. Strict code review and testing practices are crucial to mitigate this risk effectively.

Gitflow branching strategy

Gitflow is a branching model that involves the use of multiple branches to move code from development to production. Gitflow works well for teams that have scheduled release cycles and a need to define a collection of features as a release. Development is completed in individual feature branches that are merged, with approval, into a develop branch, which is used for integration. The features in this branch are considered ready for production. When all planned features have accumulated in the develop branch, a release branch is created for deployments to upper environments. This separation improves control over which changes are moving to which named environment on a defined schedule. If necessary, you can accelerate this process into a faster deployment model.

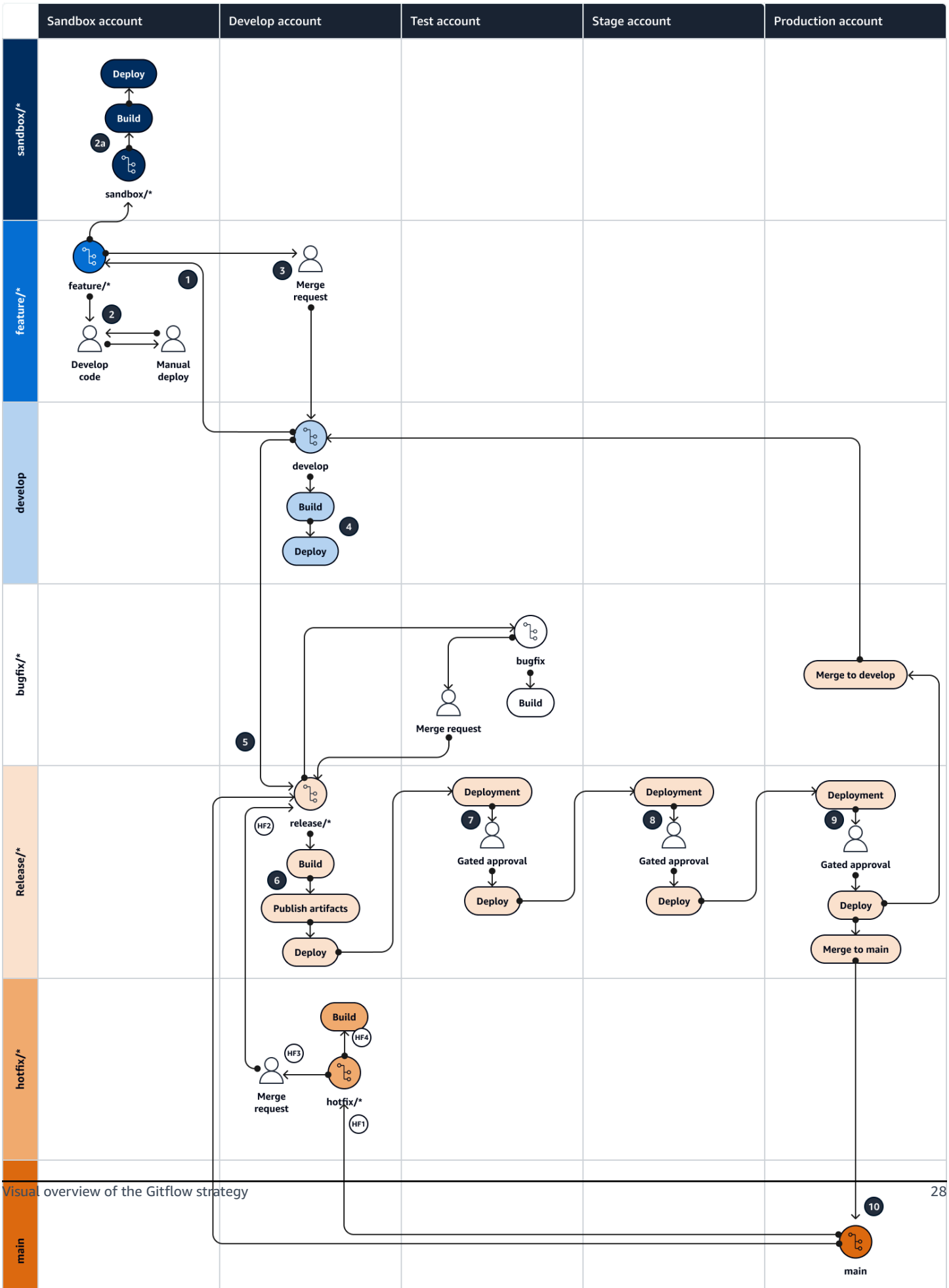
For more information about the Gitflow branching strategy, see the following resources:

- Implement a Gitflow branching strategy for multi-account DevOps environments (AWS Prescriptive Guidance)
- The original Gitflow blog (Vincent Driessen blog post)
- Gitflow workflow (Atlassian)

Topics in this section:

Visual overview of the Gitflow strategy

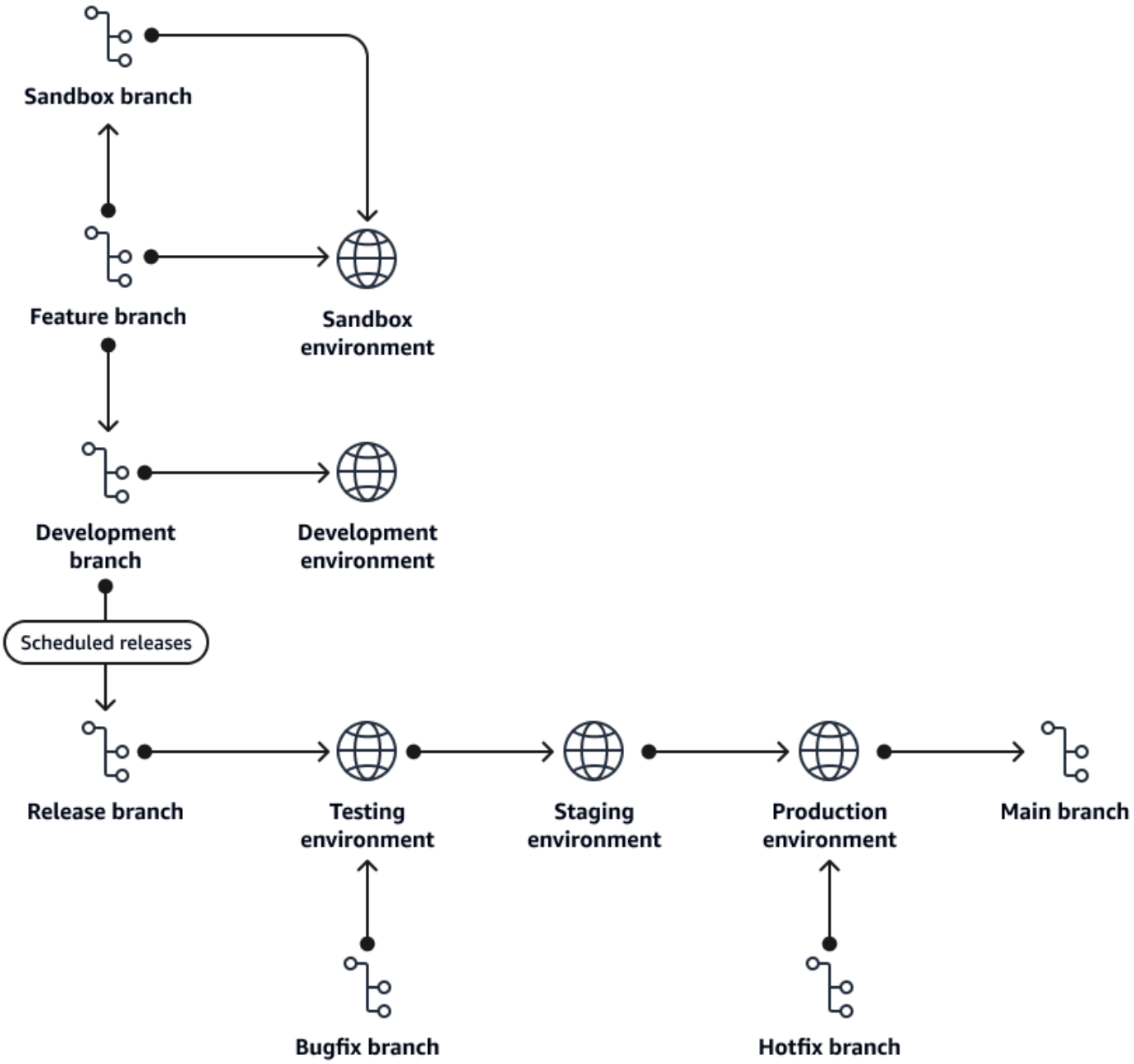
The following diagram can be used like a Punnett square to understand the Gitflow branching strategy. Line up the branches on the vertical axis with the AWS environments on the horizontal axis to determine what actions to perform in each scenario. The circled numbers guide you through the sequence of actions represented in the diagram. For more information about the activities that occur in each environment, see DevOps environments in this guide.



Visual overview of the Gitflow strategy

Branches in a Gitflow strategy

A Gitflow branching strategy commonly has the following branches.



feature branch

`Feature` branches are short-term branches where you develop features. The `feature` branch is created by branching off of the `develop` branch. Developers iterate, commit, and test code in the

`feature` branch. When the feature is complete, the developer promotes the feature. There are only two paths forward from a feature branch:

- Merge into the `sandbox` branch
- Create a merge request into the `develop` branch

Naming convention:	`feature/<story number>_<developer initials>_<descriptor>`
Naming convention example:	`feature/123456_MS_Implement_Feature_A`

sandbox branch

The `sandbox` branch is a non-standard, short-term branch for Gitflow. However, it is useful for CI/CD pipeline development. The `sandbox` branch is primarily used for the following purposes:

- Perform a full deployment to the sandbox environment by using the CI/CD pipelines rather than a manual deployment.
- Develop and test a pipeline before submitting merge requests for full testing in a lower environment, such as development or testing.

`Sandbox` branches are temporary in nature and are not meant to be long-lived. They should be deleted after the specific testing is complete.

Naming convention:	`sandbox/<story number>_<developer initials>_<descriptor>`
Naming convention example:	`sandbox/123456_MS_Test_Pipeline_Deploy`

develop branch

The `develop` branch is a long-lived branch where features are integrated, built, validated, and deployed to the development environment. All `feature` branches are merged into the `develop`

branch. Merges into the `develop` branch are completed through a merge request that requires a successful build and two developer approvals. To prevent deletion, enable branch protection on the `develop` branch.

Naming convention:	`develop`

release branch

In Gitflow, `release` branches are short-term branches. These branches are special because you can deploy them to multiple environments, embracing the build-once, deploy-many methodology. `Release` branches can target the testing, staging, or production environments. After a development team has decided to promote features into higher environments, they create a new `release` branch and use increment the version number from the previous release. At gates in each environment, deployments require manual approvals to proceed. `Release` branches should require a merge request to be changed.

After the `release` branch has deployed to production, it should be merged back into the `develop` and `main` branches to make sure that any bugfixes or hotfixes are merged back into future development efforts.

Naming convention:	`release/v{major}.{minor}`
Naming convention example:	`release/v1.0`

main branch

The `main` branch is a long-lived branch that always represents the code that is running in production. Code is merged into the `main` branch automatically from a release branch after a successful deployment from the release pipeline. To prevent deletion, enable branch protection on the `main` branch.

Naming convention:	`main`

bugfix branch

The `bugfix` branch is a short-term branch that is used to fix issues in release branches that haven't been released to production. A `bugfix` branch should only be used to promote fixes in `release` branches to the testing, staging, or production environments. A `bugfix` branch is always branched off of a `release` branch.

After the bugfix is tested, it can be promoted to the `release` branch through a merge request. Then you can push the `release` branch forward by following the standard release process.

Naming convention:	`bugfix/<ticket>_<developer initials>_<descriptor>`
Naming convention example:	`bugfix/123456_MS_Fix_Problem_A`

hotfix branch

The `hotfix` branch is a short-term branch that is used to fix issues in production. It only be used to promote fixes that must be expedited to reach the production environment. A `hotfix` branch is always branched from `main`.

After the hotfix is tested, you can promote it to production through a merge request into the `release` branch that was created from `main`. For testing, you can then push the `release` branch forward by following the standard release process.

Naming convention:	`hotfix/<ticket>_<developer initials>_<descriptor>`
Naming convention example:	`hotfix/123456_MS_Fix_Problem_A`

Advantages and disadvantages of the Gitflow strategy

The Gitflow branching strategy is well suited to larger, more distributed teams that have strict release and compliance requirements. Gitflow contributes to a predictable release cycle for the organization, and this is often preferred by larger organizations. Gitflow is also well suited for

teams that require guardrails to complete their software development lifecycle properly. This is because there are multiple opportunities for reviews and quality assurance built into the strategy. Gitflow is also well suited for teams that must maintain multiple versions of production releases simultaneously. Some disadvantages of GItflow are that is it more complex than other branching models and requires strict adherence to the pattern to complete successfully. Gitflow does not work well for organizations striving for continuous delivery due to the rigid nature of managing release branches. Gitflow release branches can be long-lived branches that can accumulate technical debt if not properly addressed in a timely manner.

Advantages

Gitflow-based development offers several advantages that can improve the development process, streamline collaboration, and enhance the overall quality of the software. The following are some of the key benefits:

- **Predictable release process** – Gitflow follows a regular and predictable release process. It is well suited to teams with regular development and release cadences.
- **Improved collaboration** – Gitflow encourages the use of `feature` and `release` branches. These two branches help teams work in parallel with minimal dependencies on each other.
- **Well suited for multiple environments** – Gitflow uses `release` branches, which can be longer-lived branches. These branches enable teams to target individual releases over a longer period of time.
- **Multiple versions in production** – If your team supports multiple versions of the software in production, Gitflow `release` branches support this requirement.
- **Built-in code quality reviews** – Gitflow requires and encourages the use of code reviews and approvals before code is promoted to another environment. This process removes friction between developers by requiring this step for all code promotions.
- **Organization benefits** – Gitflow has advantages at an organization level as well. Gitflow encourages the use of a standard release cycle, which helps the organization understand and anticipate the release schedule. Because the business now understands when new features can be delivered, there is reduced friction about timelines because there are set delivery dates.

Disadvantages

Gitflow-based development does have some disadvantages that can impact the development process and the team dynamics. The following are a few notable drawbacks:

- **Complexity** – Gitflow is a complex pattern for new teams to learn, and you must adhere to the rules of Gitflow to use this successfully.
- **Continuous deployment** – Gitflow doesn't fit a model where many deployments are released to production in a rapid fashion. This is because Gitflow requires the use of multiple branches and a strict workflow governing the `release` branch.
- **Branch management** – Gitflow uses many branches, which can become burdensome to maintain. It can be challenging to track the various branches and merge released code in order to keep the branches properly aligned with each other.
- **Technical debt** – Because Gitflow releases are typically slower than the other branching models, more features can accumulate for release, which can cause technical debt to accumulate.

Teams should carefully consider these drawbacks when deciding whether Gitflow-based development is the right approach for their project.

Next steps

This guide explains the differences between three common Git branching strategies: GitHub Flow, Gitflow, and Trunk. It describes their workflows in detail and also provides the advantages and disadvantages of each. The next steps are to choose one of these standard workflows for your organization. To implement one of these branching strategies, see the following:

- Implement a Trunk branching strategy for multi-account DevOps environments
- Implement a GitHub Flow branching strategy for multi-account DevOps environments
- Implement a Gitflow branching strategy for multi-account DevOps environments

If you are unsure where to start your team's journey to using Git and DevOps processes, we recommend picking a standard solution and testing it. Using a standard branching convention helps the team build upon existing documentation and learn what works best for them.

Don't be afraid to change your strategy if it isn't working for your organization or development teams. The needs and requirements of development teams can change over time, and there is no single, perfect solution.

Resources

This guide doesn't include training for Git; however, there are many high-quality resources available on the internet if you need this training. We recommend that you start with the Git documentation site.

The following resources can help you with your Git branching journey in the AWS Cloud.

AWS Prescriptive Guidance

- Implement a Trunk branching strategy for multi-account DevOps environments
- Implement a GitHub Flow branching strategy for multi-account DevOps environments
- Implement a Gitflow branching strategy for multi-account DevOps environments

Other AWS guidance

- AWS DevOps Guidance
- AWS Deployment Pipeline Reference Architecture
- What is DevOps?
- DevOps resources

Other resources

- Twelve-factor app methodology (12factor.net)
- Git-Secrets (GitHub)
- Gitflow
  - The original Gitflow blog (Vincent Driessen blog post)
  - Gitflow workflow (Atlassian)
  - Gitflow on GitHub: How to use Git Flow workflows with GitHub Based Repos (YouTube video)
  - Git Flow Init Example (YouTube video)
  - The Gitflow Release Branch from Start to Finish (YouTube video)
- GitHub Flow

  - GitHub Flow Quickstart (GitHub documentation)
  - Why GitHub Flow? (GitHub Flow website)
- Trunk
  - Introduction to Trunk-Based Development (Trunk Based Development website)

Contributors

Authoring

- Mike Stephens, Senior Cloud Application Architect, AWS
- Rayjan Wilson, Senior Cloud Application Architect, AWS
- Abhilash Vinod, Team Lead, Senior Cloud Application Architect, AWS

Reviewing

- Stephen DiCato, Senior Security Consultant, AWS
- Gaurav Samudra, Cloud Application Architect, AWS
- Steven Guggenheimer, Team Lead, Senior Cloud Application Architect, AWS

Technical writing

- Lilly AbouHarb, Senior Technical Writer, AWS

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	February 15, 2024

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. You migrate servers from an on-premises platform to a cloud service for the same platform. Example: Migrate a Microsoft Hyper-V application to AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

bad bot

A bot that is intended to disrupt or cause harm to individuals or organizations.

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

blue/green deployment

A deployment strategy where you create two separate but identical environments. You run the current application version in one environment (blue) and the new application version in the other environment (green). This strategy helps you quickly roll back with minimal impact.

bot

A software application that runs automated tasks over the internet and simulates human activity or interaction. Some bots are useful or beneficial, such as web crawlers that index information on the internet. Some other bots, known as *bad bots*, are intended to disrupt or cause harm to individuals or organizations.

botnet

Networks of bots that are infected by malware and are under the control of a single party, known as a *bot herder* or *bot operator*. Botnets are the best-known mechanism to scale bots and their impact.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

canary deployment

The slow and incremental release of a version to end users. When you are confident, you deploy the new version and replace the current version in its entirety.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or Bitbucket Cloud. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision (CV)

A field of AI that uses machine learning to analyze and extract information from visual formats such as digital images and videos. For example, Amazon SageMaker AI provides image processing algorithms for CV.

configuration drift

For a workload, a configuration change from the expected state. It might cause the workload to become noncompliant, and it's typically gradual and unintentional.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

CV

See computer vision.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data mesh

An architectural framework that provides distributed, decentralized data ownership with centralized management and governance.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

EDI

See electronic data interchange.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

electronic data interchange (EDI)

The automated exchange of business documents between organizations. For more information, see What is Electronic Data Interchange.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

enterprise resource planning (ERP)

A system that automates and manages key business processes (such as accounting, MES, and project management) for an enterprise.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.
- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

ERP

See enterprise resource planning.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the "2021-05-27 00:15:37" date into "2021", "May", "Thu", and "15", you can help the learning algorithm learn nuanced patterns associated with different data components.

few-shot prompting

Providing an LLM with a small number of examples that demonstrate the task and desired output before asking it to perform a similar task. This technique is an application of in-context learning, where models learn from examples (*shots*) that are embedded in prompts. Few-shot prompting can be effective for tasks that require specific formatting, reasoning, or domain knowledge. See also zero-shot prompting.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

FM

See foundation model.

foundation model (FM)

A large deep-learning neural network that has been training on massive datasets of generalized and unlabeled data. FMs are capable of performing a wide variety of general tasks, such as understanding language, generating text and images, and conversing in natural language. For more information, see What are Foundation Models.

G

generative AI

A subset of AI models that have been trained on large amounts of data and that can use a simple text prompt to create new content and artifacts, such as images, videos, text, and audio. For more information, see What is Generative AI.

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

golden image

A snapshot of a system or software that is used as a template to deploy new instances of that system or software. For example, in manufacturing, a golden image can be used to provision software on multiple devices and helps improve speed, scalability, and productivity in device manufacturing operations.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction

of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries. *Detective guardrails* detect policy violations and compliance issues, and generate alerts for remediation. They are implemented by using AWS Config, AWS Security Hub CSPM, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

holdout data

A portion of historical, labeled data that is withheld from a dataset that is used to train a machine learning model. You can use holdout data to evaluate the model performance by comparing the model predictions against the holdout data.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

Industry 4.0

A term that was introduced by Klaus Schwab in 2016 to refer to the modernization of manufacturing processes through advances in connectivity, real-time data, automation, analytics, and AI/ML.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

See Internet of Things.

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

See IT information library.

ITSM

See IT service management.

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large language model (LLM)

A deep learning AI model that is pretrained on a vast amount of data. An LLM can perform multiple tasks, such as answering questions, summarizing documents, translating text into other languages, and completing sentences. For more information, see What are LLMs.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

LLM

See large language model.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

malware

Software that is designed to compromise computer security or privacy. Malware might disrupt computer systems, leak sensitive information, or gain unauthorized access. Examples of malware include viruses, worms, ransomware, Trojan horses, spyware, and keyloggers.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

manufacturing execution system (MES)

A software system for tracking, monitoring, documenting, and controlling production processes that convert raw materials to finished products on the shop floor.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

MES

See manufacturing execution system.

Message Queuing Telemetry Transport (MQTT)

A lightweight, machine-to-machine (M2M) communication protocol, based on the publish/subscribe pattern, for resource-constrained IoT devices.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed,

and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners, migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO

comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can

use a microservices architecture. For more information, see Decomposing monoliths into microservices.

MPA

See Migration Portfolio Assessment.

MQTT

See Message Queuing Telemetry Transport.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

OPC-UA

See Open Process Communications - Unified Architecture.

Open Process Communications - Unified Architecture (OPC-UA)

A machine-to-machine (M2M) communication protocol for industrial automation. OPC-UA provides an interoperability standard with data encryption, authentication, and authorization schemes.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operational technology (OT)

Hardware and software systems that work with the physical environment to control industrial operations, equipment, and infrastructure. In manufacturing, the integration of OT and information technology (IT) systems is a key focus for Industry 4.0 transformations.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the

organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic PUT and DELETE requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

OT

See operational technology.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

PLC

See programmable logic controller.

PLM

See product lifecycle management.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more

easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

privacy by design

A system engineering approach that takes privacy into account through the whole development process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the

AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

product lifecycle management (PLM)

The management of data and processes for a product throughout its entire lifecycle, from design, development, and launch, through growth and maturity, to decline and removal.

production environment

See environment.

programmable logic controller (PLC)

In manufacturing, a highly reliable, adaptable computer that monitors machines and automates manufacturing processes.

prompt chaining

Using the output of one LLM prompt as the input for the next prompt to generate better responses. This technique is used to break down a complex task into subtasks, or to iteratively refine or expand a preliminary response. It helps improve the accuracy and relevance of a model's responses and allows for more granular, personalized results.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

publish/subscribe (pub/sub)

A pattern that enables asynchronous communications among microservices to improve scalability and responsiveness. For example, in a microservices-based MES, a microservice can publish event messages to a channel that other microservices can subscribe to. The system can add new microservices without changing the publishing service.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

RAG

See Retrieval Augmented Generation.

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Specify which AWS Regions your account can use.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resiliency

An application's ability to resist or recover from disruptions. High availability and disaster recovery are common considerations when planning for resiliency in the AWS Cloud. For more information, see AWS Cloud Resilience.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

Retrieval Augmented Generation (RAG)

A generative AI technology in which an LLM references an authoritative data source that is outside of its training data sources before generating a response. For example, a RAG model might perform a semantic search of an organization's knowledge base or custom data. For more information, see What is RAG.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCADA

See supervisory control and data acquisition.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata. The secret value can be binary, a single string, or multiple strings. For more information, see What's in a Secrets Manager secret? in the Secrets Manager documentation.

security by design

A system engineering approach that takes security into account through the whole development process.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization's capabilities and services, improves developer productivity, and supports rapid

innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

supervisory control and data acquisition (SCADA)

In manufacturing, a system that uses hardware and software to monitor physical assets and production operations.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

system prompt

A technique for providing context, instructions, or guidelines to an LLM to direct its behavior. System prompts help set context and establish rules for interactions with users.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zero-shot prompting

Providing an LLM with instructions for performing a task but no examples (*shots*) that can help guide it. The LLM must use its pre-trained knowledge to handle the task. The effectiveness of zero-shot prompting depends on the complexity of the task and the quality of the prompt. See also few-shot prompting.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.