AWS Well-Architected

Reducing the Scope of Impact with Cell-Based Architecture

Reducing the Scope of Impact with Cell-Based Architecture: AWS Well-Architected

Table of Contents

Abstract and introduction i
Introduction 1
Are you Well-Architected? 2
Shared Responsibility Model for Resiliency 3
What is a cell-based architecture? 4
A typical workload 5
A workload with cell-based architecture 7
Why use a cell-based architecture? 9
Scale-out over scale-up 9
Lower scope of impact 10
Higher scalability or cells as a unit scale 10
Higher testability 12
Higher mean time between failure (MTBF) 12
Lower mean time to recovery (MTTR) 12
Higher availability 12
More control over the impact of deployments and rollbacks 13
When to use a cell-based architecture? 14
Implementing a cell-based architecture 15
Control plane and data plane 15
Cell design 16
Multi-AZ cells 17
Single-AZ cells 18
Should a Single-AZ cell fail over if an AZ becomes unavailable? 20
Cell partition 22
Full mapping 23
Prefix and range-based mapping 24
Naïve modulo mapping or fixed partition number 25
Consistent hashing 26
A warning for all mapping approaches 27
Cell routing 27
Using Amazon Route 53 29
Using Amazon API Gateway as cell router 30
Using a compute layer for cell mapping as cell router 31
Routing no-HTTP requests 32

About resilience of the cell router ... 34
Cell sizing ... 34
Define your scaling dimensions ... 37
Know and respect your cell's limit ... 39
Cell placement ... 39
Cell migration ... 41
Cell deployment ... 41
Cell observability ... 44
Best practices ... 47
Your current instance/stack is your cell zero ... 47
Start with multiple cells from day one ... 47
Start with a cell migration mechanism from day one ... 47
Perform a failure mode analysis of your cell ... 47
Conclusion ... 48
FAQ ... 49
What about shuffle-sharding? ... 49
Contributors ... 50
Further reading ... 51
Videos ... 51
Document history ... 52
Notices ... 53
AWS Glossary ... 54

Reducing the Scope of Impact with Cell-Based Architecture

Publication date: **September 20, 2023** (*Document history*)

Today, modern organizations face an increasing number of challenges related to resiliency, be they scalability or availability, especially when customer expectations shift to an *always on, always available* mentality. More and more, we have remote teams and complex distributed systems, along with the growing need for frequent launches and an acceleration of teams, processes, and systems moving from a centralized model to a distributed model. All of this means that an organization and its systems need to be more resilient than ever.

With the increasing use of cloud computing, sharing resources efficiently has become easier. The development of multi-tenant applications is increasing exponentially, but although the use of the cloud is more understood and customers are aware that they are in a multi-tenant environment, what they still want is the experience of a single-tenant environment.

This guidance aims to demonstrate how to increase the resilience of critical applications, bringing the same fault isolation concepts that AWS applies in its Availability Zones and Regions to the level of your workload architecture. It expands one of the best practices from the AWS Well-Architected Framework, Use bulkhead architectures to limit scope of impact, to help you reduce the effect of failures to a limited number of components.

Introduction

Resilience is the ability for workloads—a collection of resources and code that delivers business value, such as a customer-facing application or backend process—to respond and quickly recover from failures.

At AWS, we strive to build, operate, and deliver extremely resilient services, and build recovery mechanisms and mitigations, while keeping in mind that everything fails all the time. AWS provides different isolation boundaries, such as Availability Zones (AZs), AWS Regions, control planes, and data planes. These are not the only mechanisms we use. For more than a decade, our service teams have used cell-based architecture to build more resilient and scalable services.

Our service teams have a track record of developing services on a global scale with high availability and one of the things that contributes to this high availability and resiliency is the use of cell-based architecture.

Every organization is at a different point in their resilience journey—some are just beginning, while others might be more advanced and may require extreme levels of resilience in their applications. For these types of applications, this guidance can help you increase the resiliency of your applications and increase the trust of your services with your customers. Cell-based architecture can give your workload more fault isolation, predictability, and testability. These are fundamental properties for workloads that need extreme levels of resilience.

Are you Well-Architected?

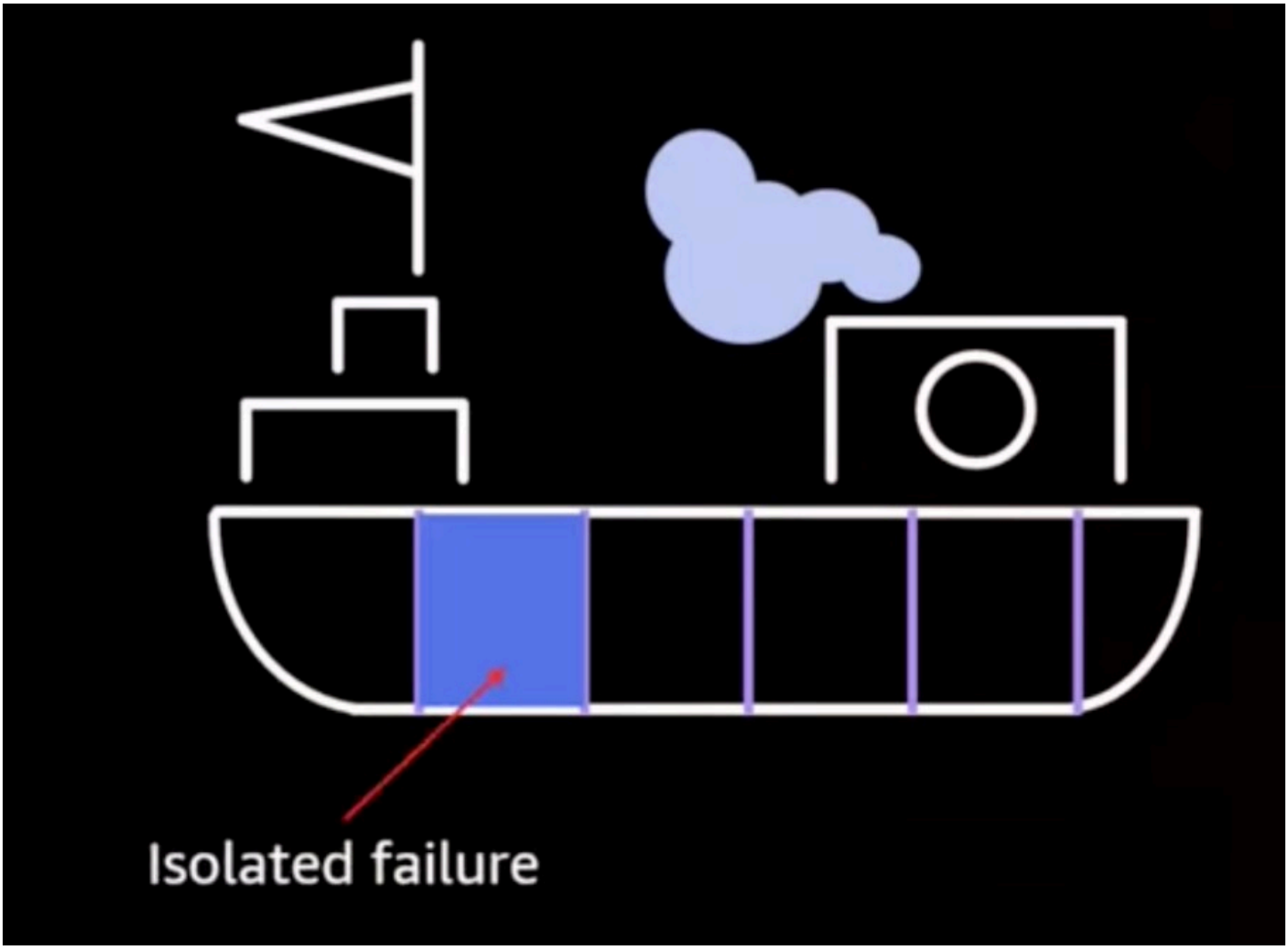
The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems. Using the AWS Well-Architected Tool, available at no charge in the AWS Management Console, you can review your workloads against these best practices by answering a set of questions for each pillar.

Shared Responsibility Model for Resiliency

In the Shared Responsibility Model for Resiliency, you can see more details about AWS responsibility *Resiliency **of** the Cloud* and the customer responsibility *Resiliency **in** the Cloud*. This paper will show you how to design your workload architecture that is part of Resilience in the Cloud (your responsibility) in order to increase its resiliency. It is important that you understand how the cell-based architecture within the workload architecture operates under this shared model. With it, you can add another level of fault isolation, going beyond the Availability Zones and Regions that AWS provides.

What is a cell-based architecture?

A cell-based architecture comes from the concept of a bulkhead in a ship, where vertical partition walls subdivide the ship's interior into self-contained, watertight compartments. Bulkheads reduce the extent of seawater flooding in case of damage and provide additional stiffness to the hull girder.



Isolating failures using bulkheads

On a ship, bulkheads ensure that a hull breach is contained within one section of the ship. In complex systems, this pattern is often replicated to allow fault isolation. *Fault isolated boundaries* restrict the effect of a failure within a workload to a limited number of components. Components outside of the boundary are unaffected by the failure.

Using multiple fault isolated boundaries, you can limit the impact on your workload. When provisioning a new customer or tenant, or applying a workload change, you can do this gradually, compartment by compartment, or in other words, isolation boundary by isolation boundary. This way, when a failure occurs, a smaller number of customers or resources will be impacted. On AWS, customers can use multiple Availability Zones and AWS Regions to provide fault isolation, but the concept of fault isolation can be extended to your workload's architecture as well.

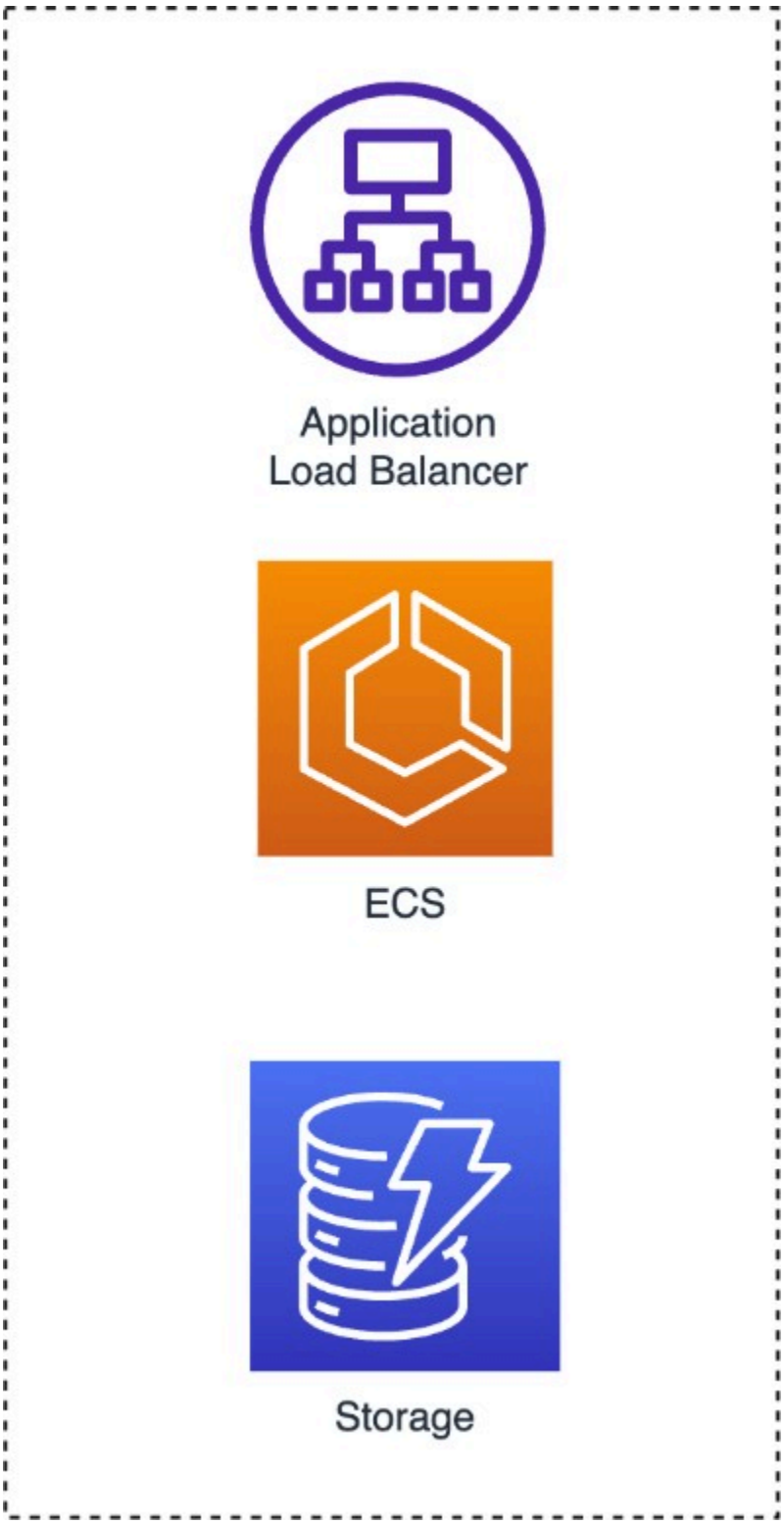
The overall workload is partitioned by a partition key. This key needs to align with the *grain* of the service, or the natural way that a service's workload can be subdivided with minimal cross-cell interactions. Examples of partition keys are customer ID, resource ID, or any other parameter easily accessible in most API calls. A cell routing layer distributes requests to individual cells based on the partition key and presents a single endpoint to clients.

A cell-based architecture uses multiple isolated instances of a workload, where each instance is known as a *cell*. Each cell is independent, does not share state with other cells, and handles a subset of the overall workload requests. This reduces the potential impact of a failure, such as a bad software update, to an individual cell and the requests that it's processing. If a workload uses 10 cells to service 100 requests, when a failure occurs in one cell, 90% of the overall requests would be unaffected by the failure.

With cell-based architectures, many common types of failure are contained within the cell itself, providing additional fault isolation. These fault boundaries can provide resilience against failure types that otherwise are hard to contain, such as unsuccessful code deployments or requests that are corrupted or invoke a specific failure mode (also known as *poison pill requests*).

A typical workload

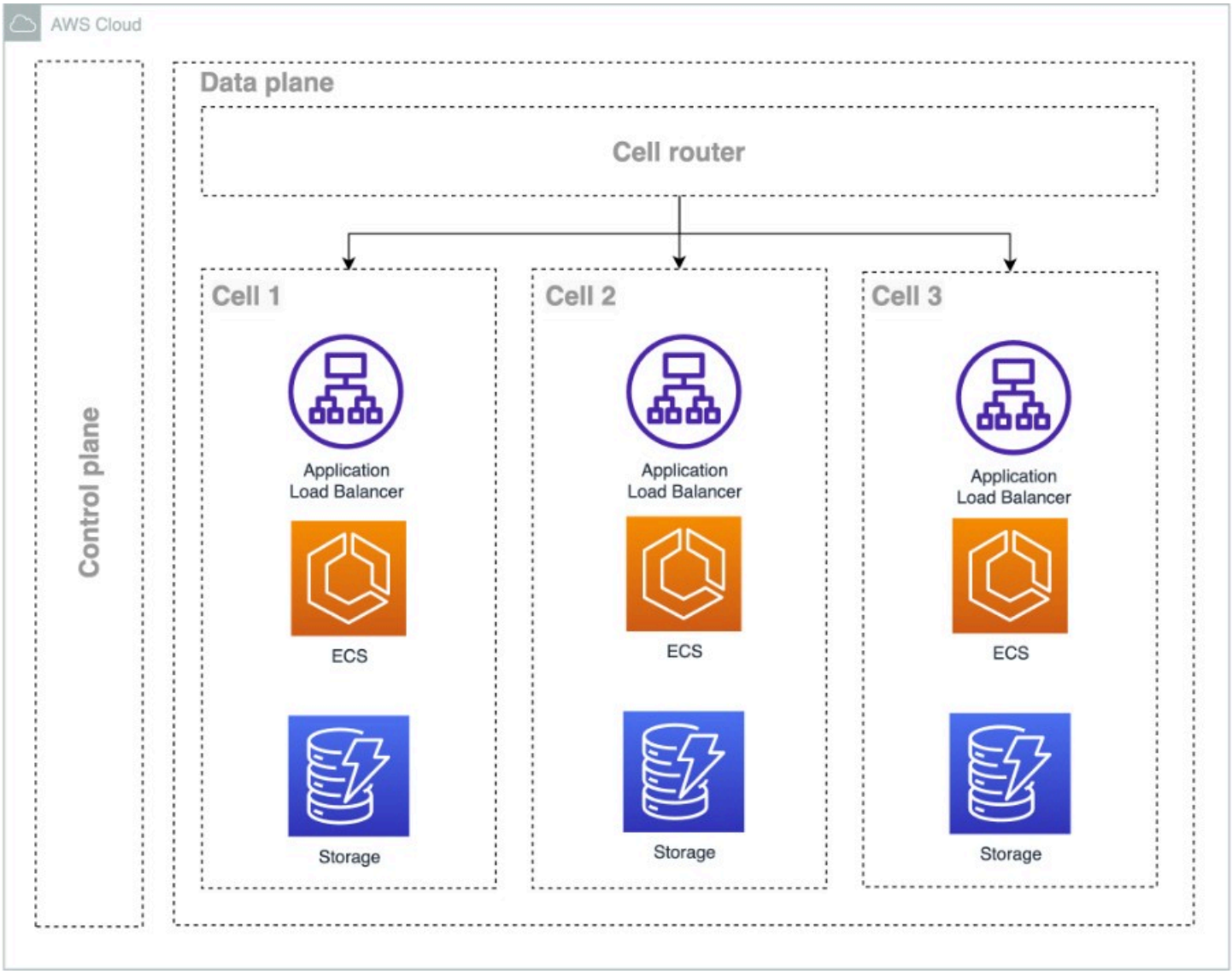
To make it clearer, in the following diagram, we have a typical application divided into three layers. In this context, this application would be serving requests from 100% of clients. In the event of a failure, or a change in the application, 100% of customers would be impacted.



A typical workload

A workload with cell-based architecture

Rather than build out services as single-image systems, we propose a different approach: break your services down internally into cells and build thin layers to route traffic to the right cells. This type of architecture can be zonal, regional, or global.



A cell-based architecture

The cell-based architecture has the following components, which will be further explored later in this guidance:

- **Cell router** — We also refer to this layer as the *thinnest possible layer*, with the responsibility of routing requests to the right cell, and only that.

- **Cell** — A complete workload, with everything needed to operate independently.
- **Control plane** — Responsible for administration tasks, such as provisioning cells, de-provisioning cells, and migrating cell customers.

Building a cell-based architecture doesn't necessarily mean having to double, triple, or more your application's infrastructure. It might be that your application has 30 hosts, and in a cell-based architecture it has the same 30 hosts, but with a cell router and with tasks that are distributed or grouped between cells.

Why use a cell-based architecture?

These are the main advantages of a cell-based architecture.

Scale-out over scale-up

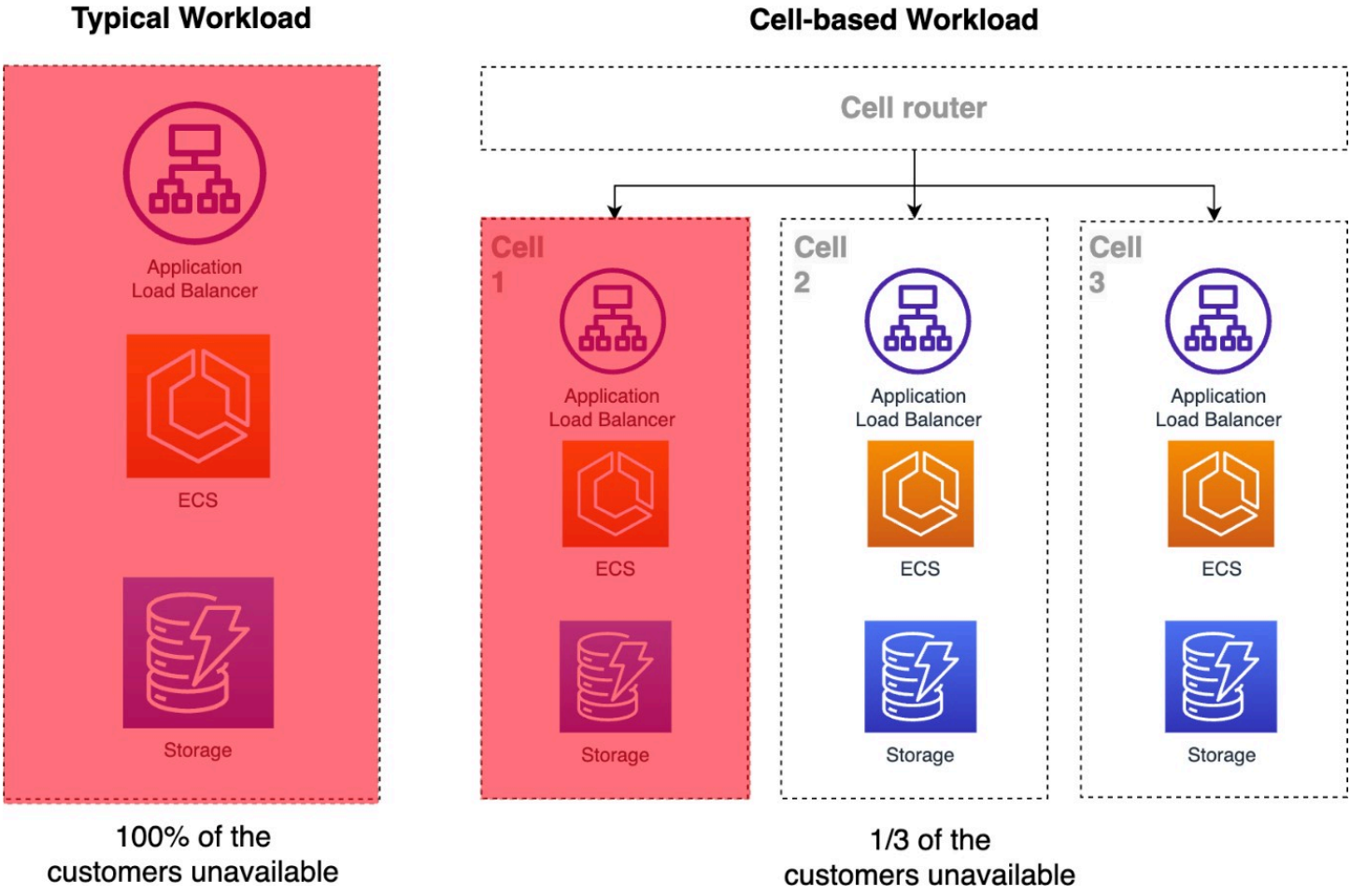
Scaling up, or accommodating growth by increasing the size of a system's component (such as a database, server, or subsystem) is a natural and straightforward way to scale. Scaling out, on the other hand, accommodates growth by increasing the *number* of system components (such as databases, servers, and subsystems), and dividing the workload such that the load on any component stays bounded over time despite the overall increase in workload. The task of dividing the workload can make scaling out more challenging than scaling up, particularly for stateful systems, but has well-understood benefits:

- **Workload isolation:** Dividing the workload across components means workloads are isolated from each other. This provides failure containment and narrows the impact of issues, such as deployment failures, poison pills, misbehaving clients, data corruption, and operational mistakes.
- **Maximally-sized components:** Accommodating growth by increasing the number of components rather than increasing component size means that the size of each component can be capped to a maximum size. This reduces the risk of surprises from non-linear scaling factors and hidden contention points present in scale-up systems.
- **Not too big to test:** With maximally-sized components, the components are no longer too big to test. These components can be stress tested and pushed past their breaking point to understand their safe operating margin. This approach does not address testing the overall scaled out system composed of these components, but if the majority of the complexity and risk of the system sits in stress-tested components (and it should), the level of test coverage and confidence should be significantly higher.

Cells change our approach from scaling up to scaling out. Each cell, a complete independent instance of the service, has a fixed maximum size. Beyond the size of a single cell, regions grow by adding more cells. This change in design doesn't change the customer experience of your services. Customers can continue to access the services as they do today.

Lower scope of impact

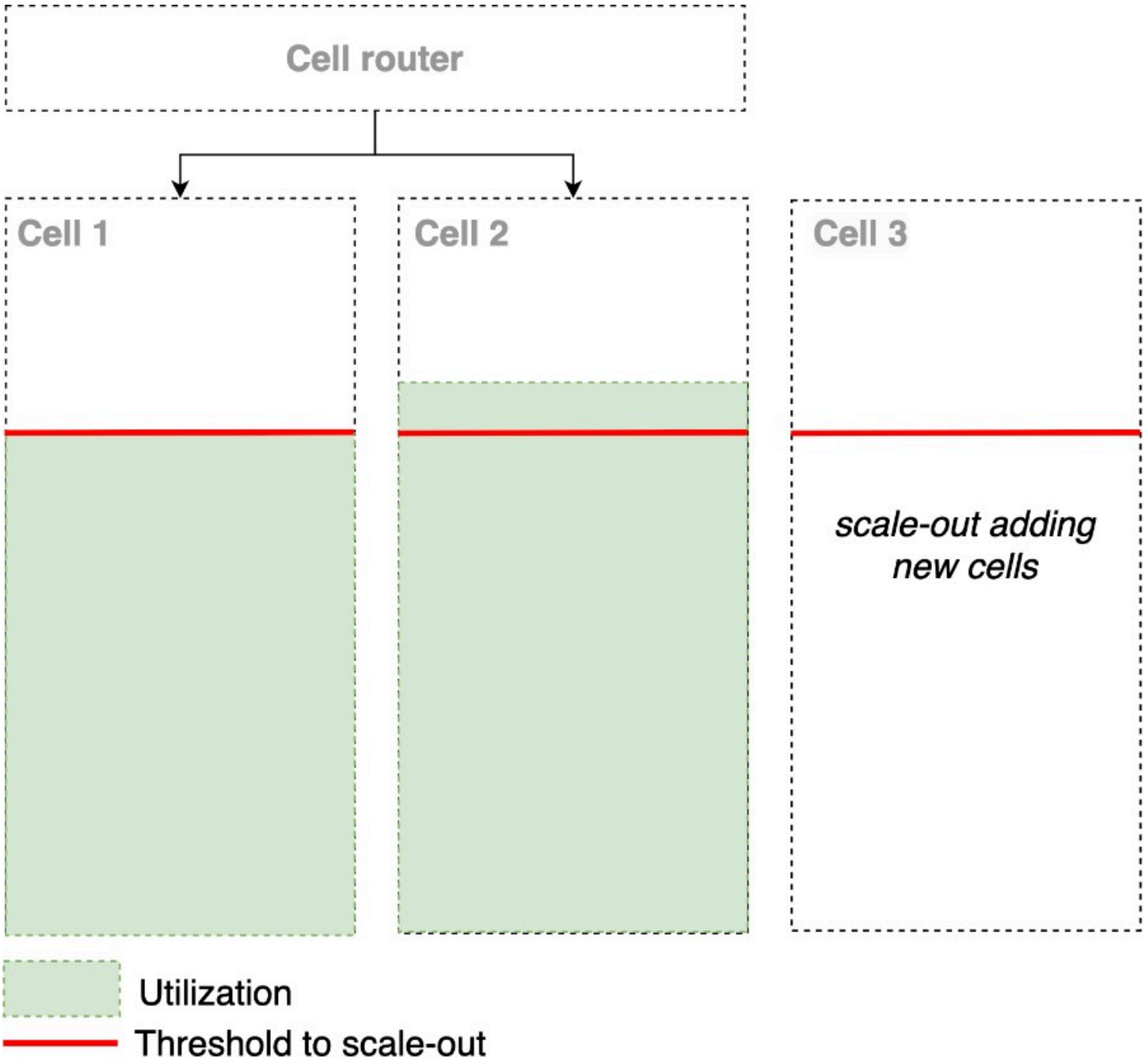
Breaking a service up into multiple cells reduces the scope of impact. Cells represent bulkheaded units that provide containment for many common failure scenarios. When properly isolated from each other, cells have failure containment similar to what we see with Regions. It's highly unlikely for a service outage to span multiple Regions. It should be similarly unlikely for a service outage to span multiple cells.



Cell-based architectures can reduce scope of impact

Higher scalability or cells as a unit scale

As recommended in Manage service quotas and constraints in the Well-Architected Framework, for your workloads, defining, testing, and managing the limits and capacity of a cell is also essential. Knowing and monitoring this capacity, it's possible to define limits, and scale your workload by adding new cells to your architecture, thus scaling it out.



Scale-out with multiple cells

Cell-based architectures scale-out rather than scale-up, and are inherently more scalable. This is because when scaling up, you can reach the resource limits of a particular service, instance, or AWS account. However, scaling out your workload within an Availability Zone, Region, and AWS account, you can avoid reaching the limits of a specific service or resource. When building cells with a fixed size and known and testable limits, it is possible to add new cells that are in accordance with the limits of the same cited resources.

Higher testability

Testing distributed systems is a challenging task and is amplified as the system grows. The capped size of cells allows for well-understood and testable maximum scale behavior. It is easier to test cells as compared to bulk services since cells have a limited size. It is impractical for cost reasons for large-scale services to regularly simulate the entire workload of all their tenants, but it is reasonable to simulate the largest workload that can fit into a cell, which should match the largest workload that a single customer can send to your application.

Higher mean time between failure (MTBF)

Not only is the scope of impact of an outage reduced with cells, but so is the probability of an outage. Cells have a consistent capped size that is regularly tested and operated, eliminating the *every day is a new adventure* dynamic.

As in day-to-day operations, your customers are distributed among the cells and a problem can be identified locally. The same goes for new versions of applications that can only be applied to a small number of cells (up to just one) and when a failure is identified, rollback.

With your customers spread across cells, for example, 10, you now have 10% of your customers in each cell. This, added to a gradual deployment strategy that we will describe later in this guidance, allows you to better manage system changes, and contain the scope of failures such as code or traffic spikes in some cells while others remain stable and unaffected, thus increasing the average time between application failures.

Lower mean time to recovery (MTTR)

Cells are also easier to recover, because they limit the number of hosts that need to be analyzed and touched for problem diagnosis and the deployment of emergency code and configuration. The predictability of size and scale that cells bring also make recovery more predictable in the event of a failure.

Higher availability

A natural conclusion is that cell-based architectures should have the same overall availability as monolithic systems, because a system with n cells will have n times as many failure events, but

each with $1/n^{th}$ of the impact. But the higher MTBF and lower MTTR afforded by cells means fewer shorter failures events per cell, and higher overall availability.

There is also the availability defined by:

```
#successful requests / #total requests
```

With cells, you can minimize the amount of time the numerator is zero.

More control over the impact of deployments and rollbacks

Like one-box and Single-AZ deployments, cells provide another dimension in which to phase deployments and reduce scope of impact from problematic deployments. Further, the first cell deployed to in a phased cell deployment can be a canary cell, and each cell can have its own canary with synthetic and other non-critical workloads to further reduce the impact of a failed deployment.

Applications that do not use cell-based architecture can also benefit from strategies such as canary deployment. But what the cells bring is the possibility of a combination of a canary deployment strategy being applied in a context of even lesser impact.

When to use a cell-based architecture?

There are applications that serve such a large number of customers that interruption of all customers is unacceptable, either for reputation, or financial reasons, where every unavailable second has a big impact. Workloads that have the following characteristics can benefit from a cell-based architecture:

- Applications where any downtime can have a huge negative impact on customers.
- FSI customers with workloads critical to economic stability.
- Ultra-scale systems that are too big/critical to fail.
- Less than 5 seconds of Recovery Point Objective (RPO).
- Less than 30 seconds of Recovery Time Objective (RTO).
- Multi-tenant services where some tenants require fully dedicated tenancy (meaning, their own dedicated cell).

A question to ponder regarding your workload is this: *"Is it better for 100% of customers to experience a 5% failure rate, or 5% of customers to experience a 100% failure rate?"*

Cell-based architecture will not be a good choice for all your workloads, but it can bring great benefits for some your most critical workloads.

Implementing a cell-based architecture is not a simple task, among the disadvantages are:

- Increase in the complexity of the architecture due to the redundancy of infrastructure and components.
- High cost of infrastructure and services, although utilization based fee structures like Amazon EC2 Reserved Instances (RIs) and saving plans help close this delta.
- Requires specialized operational tools and practices to operate these multiple replicas (cells) of the workload.
- Necessity to invest in a cell routing layer.

Implementing a cell-based architecture

When designing a cell-based architecture, there are several design considerations. This isn't a conclusive description, but some of the learnings that AWS service teams have learned from implementing cell-based architecture.

Topics

- Control plane and data plane
- Cell design
- Cell partition
- Cell routing
- Cell sizing
- Cell placement
- Cell migration
- Cell deployment
- Cell observability

Control plane and data plane

AWS separates most services into the concepts of *control plane* and *data plane*. These terms come from the world of networking, specifically routers. The router's data plane, which is its main function, is moving packets around based on rules. But the routing policies have to be created and distributed from somewhere, and that's where the control plane comes in.

Control planes for your cell-based architecture provide the administrative APIs used to provision, move, migrate, update, remove, deploy, and monitor cells, among others. The data plane is what provides the primary function of the service together with cell router.

To understand the relationship between the control plane and data plane in a cell-based architecture, imagine that you have five cells and the number of users starts growing. Your control plane is responsible for provisioning a new cell and letting the router to know where traffic needs to be sent to. After that, both the router and the cell will be just performing the work they're supposed to (data plane).

Another important aspect to consider here is the static stability as recommended in the Well-Architected Framework, REL11-BP04 Rely on the data plane and not the control plane during recovery. In a statically stable design, the overall system keeps working even when a dependency becomes impaired. For the cell-based context, it would be the data plane to continue operating even if the control plane is down, or even if some availability zone is also down.

Control planes are statistically more likely to fail than data planes. Although the data plane typically depends on data that arrives from the control plane, the data plane maintains its existing state and continues working even in the face of control plane impairment. Data plane access to resources, once provisioned, has no dependency on the control plane, and therefore is not affected by any control plane impairment. In other words, even if the ability to create, modify, or delete resources is impaired, existing resources remain available. You can implement different patterns to be statically stable against different types of dependency failures depending on your cell design strategy.

In addition, we can say that control planes are designed to fail rather than corrupt or provide incorrect information (CP in the CAP theorem), while data planes generally prefer AP in the CAP theorem (Data planes try their best to remain available, even if they depend on stale information for decisions.) An example would be Using a compute layer like Amazon EC2, Amazon ECS or Amazon EKS and Amazon S3 for cell mapping as cell router described later in this paper. Routes to cells are loaded into memory from an S3 bucket. Even if the control plane, Amazon S3 or a zone is unavailable, the router is still able to direct traffic to the cells.

Cell design

A cell is an instance of your complete workload, with everything needed to operate independently. As an example, consider an application that is comprised by an Application Load Balancer, some EC2 instances, and an Amazon RDS database. In this case, all components are in your cell, for example, `cell 1`. If you need another cell, `cell 2`, you will have to create another deployment of these three components.

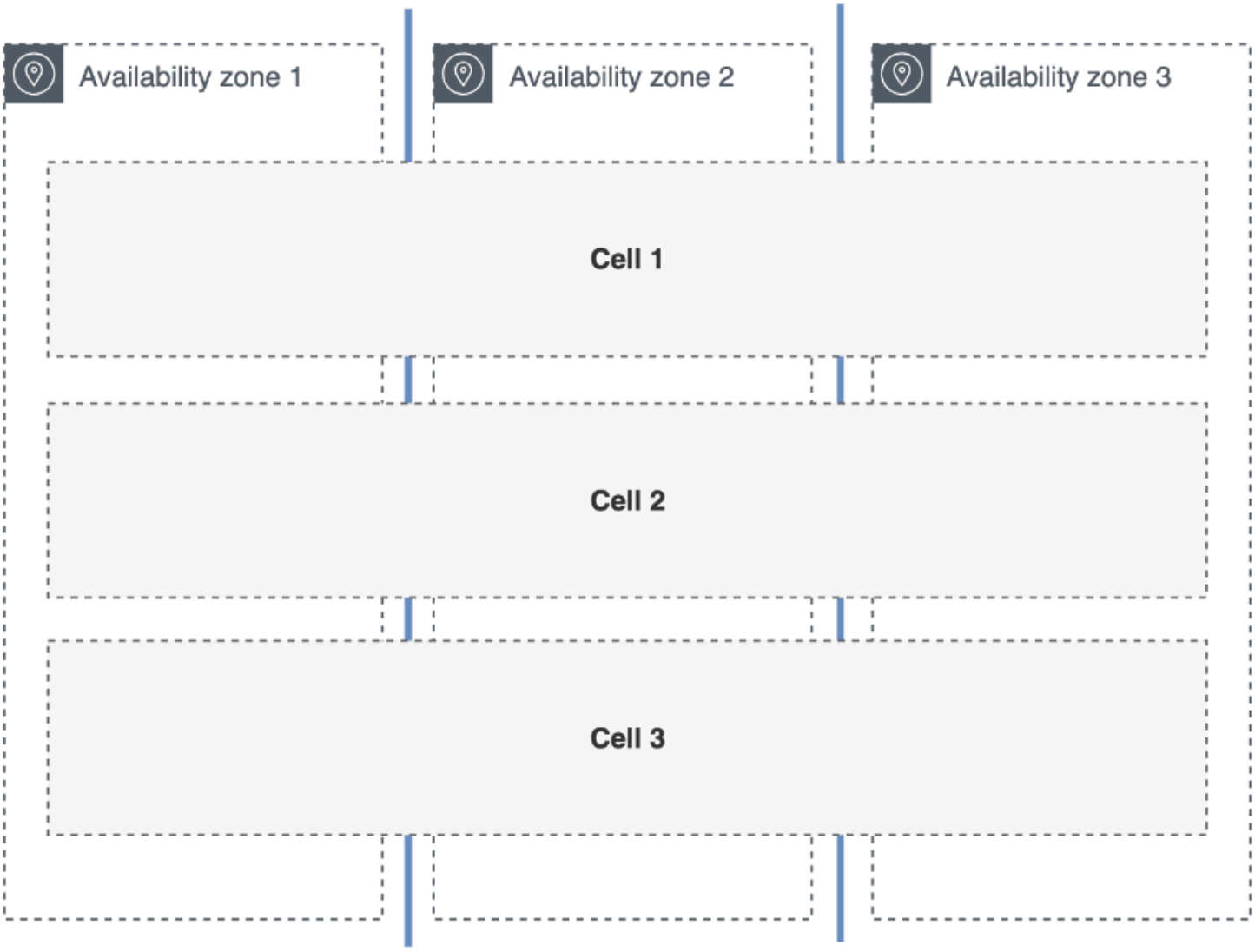
How you define your cell boundaries has a profound impact on the resiliency, cost, and architecture of your workload. We are going to describe some ways to define your cell strategy with its pros and cons.

In an ideal world, a cell is independent, is unaware of other cells, and does not share its state with other cells. Cells should have no dependency on each other at all (that is, no cross-cell API calls, no shared resources like databases or S3 buckets.) Even the use of separate AWS accounts

is encouraged. However, depending on your workload, it is not always possible to maintain these characteristics. Cross-cell dependencies can quickly eliminate the benefits of a cellular architecture, so try to do this as little as possible, or only at specific transitory times.

Multi-AZ cells

A way to reduce the scope of impact of a service is to introduce Multi-AZ cells. In this approach, cells are Multi-AZ or Regional, and can take advantage of all the resiliency and availability of Regional and Multi-AZ services already offered by AWS, thus abstracting the great complexity that this management requires. Each replica of your workload (cell) will continue to running even if an AZ is unavailable for the subset of clients or traffic you have defined.



Multi-AZ cells

Advantages of this approach:

- Wider use of serverless services that already are Regional.
- It is easier to a cell be self-resilience using serverless or managed services with a Multi-AZ strategy, without share state with external components.

Disadvantages of this approach:

- Less control over an AZ failure, particularly gray failures, where some components or services are unstable. In this case, evacuate one isolation zone, where that zone can be an AZ or a Region may not solve the problem.

Single-AZ cells

Another way to divide your workload into cells is to break the service down to follow Availability Zone (AZ) boundaries. This approach is suitable for services that expose Availability Zones directly as a failure unit. Like Amazon EC2, for example, asks to choose an AZ for their instances, and encourages to build their systems to tolerate the failure of a single Availability Zone.

Single-AZ cells

Advantages of this approach:

- It is possible to accurately detect in which AZ a problem is occurring and take mitigation actions for it.

Disadvantages of this approach:

- Requires three cell routers, and requires clients to chose the correct zonal endpoint.
- Require using services that have the AZ scope in its configuration

- It requires additional disaster recovery mechanisms such as active-passive or active-active to maintain cell resiliency. Cell state needs to be replicate to another, which in turn can break the cell concept.

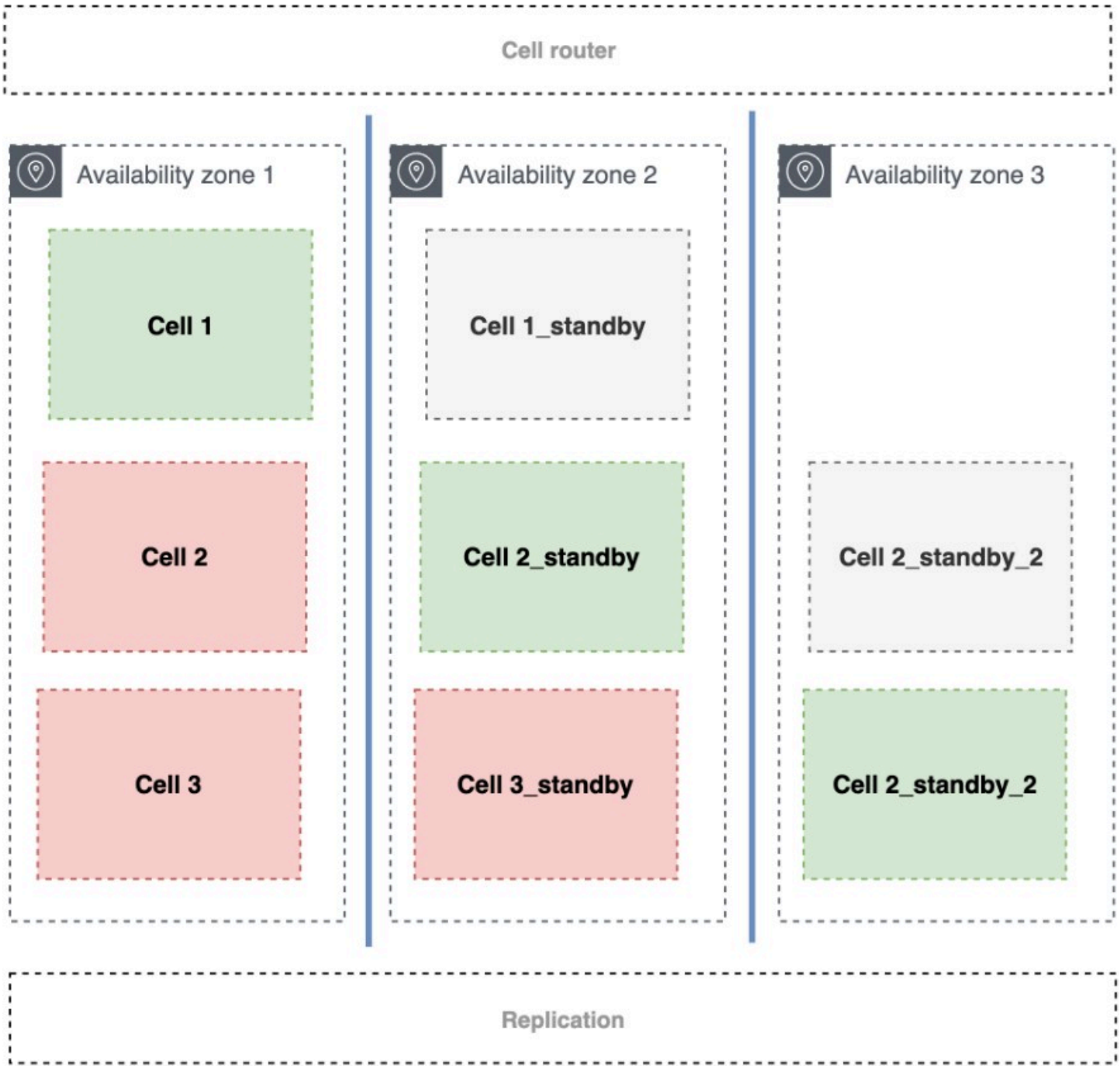
Should a Single-AZ cell fail over if an AZ becomes unavailable? Or in the event of a gray failure?

Cells are implemented primarily to limit the scope of a failure's impact. The failures this mainly helps with are those that cause cascading failures. Mainly, they are excessive load of resources and deployments with problems or bugs. This means that the cells isolate failure from deficiencies caused by single or multiple client load or an incorrect deployment in that cell. Cells were not intended to mitigate dependency failures or single points of failure. Therefore, they were not designed as failover domains.

Even so, we often get this question and the following scenario. As in the previous example, what happens if an AZ becomes unavailable? We will have four cells down, with their respective customers impacted. In the proportion of the example, it would be a third of your customers having your services unavailable. This will vary according to your defined cell size.

With this scenario you have a few options:

- Cells are implemented primarily to limit the scope of a failure's impact and not as failover domains. Soon the fault will be tolerated according to its isolation scope.
- Use more traditional Disaster Recovery mechanisms combined with cells that are Single-AZ.



Impact of an AZ failure

In this scenario, each cell that is Single-AZ has one or more replicas of itself in other Availability Zones. But as the word itself indicates, a replica demands a replication layer. This replication layer can vary according to the type of stateful component your cell is using. It could be an Amazon RDS data service, it could be a DynamoDB database, Amazon ElastiCache, an event service like Kinesis or Amazon SQS. Each service will have a different replication strategy as well as the DR approach

you take as pilot-light, warm-standby or active-active, as described in the Disaster Recovery of Workloads on AWS whitepaper.

In the event that a zone becomes unavailable, the same cell replicated in another AZ can take over the work and continue processing customer traffic. But this can also happen in the case of gray failures in one or more components, when an evacuation of a cell is more beneficial than living with the failure until it is detected and corrected.

This second approach is more complex and driven by a much higher cost in terms of infrastructure. You are responsible for creating the mechanisms that will ensure high availability for the cell at a higher layer, as well as data replication issues to mitigate the cases where a zone might fail. If your workload is not offering a service that has the scope of execution inside an Availability Zone to your customers, the Multi-AZ cell is a better approach to consider.

Cell partition

Cell partition is related to how you will divide your traffic between cells, since each cell is an independent replica of your workload. This division is done using a partition key, which can be simple or composite according to your workload requirements.

Partition keys must be chosen to match the *grain* of the service, or the natural ways that a service's workload can be subdivided with minimal cross-grain interactions. A good partition key is one that is easily accessible in most API calls, either as a direct parameter or a direct transformation of a parameter.

A key consideration in selecting a partition key is the maximum cell size requirements. `CustomerID` might seem like a reasonable candidate partition key for many use cases, but thought needs to go into how to handle really large customers. It is unlikely you want to limit the degree to which a customer can adopt your service. For example, if you choose the `CustomerID` for your partition key and a single customer of yours becomes so big that it doesn't fit into a single cell anymore, but it needs to be allocated across two cells. A good strategy is to define a second dimension more aligned with your type of business to be part of the partition key along with the customerId of the customer in this example.

Some service interactions might go against the grain of the partition key, or cause the workload to span multiple cells (for example, scatter-gather). These are inevitable and need to be accommodated, but should represent the minority of the service's workload. One approach for this is that instead of letting the cells talk directly to each other, any cross-cell calls have to go back through the normal cell router.

There are a variety of partitioning algorithms that can be used to map keys to cells. Regardless of algorithm, there needs to be:

- A mechanism to serve or distribute state used by these algorithms,
- Accommodations for gracefully handling migration when cells are added and removed.

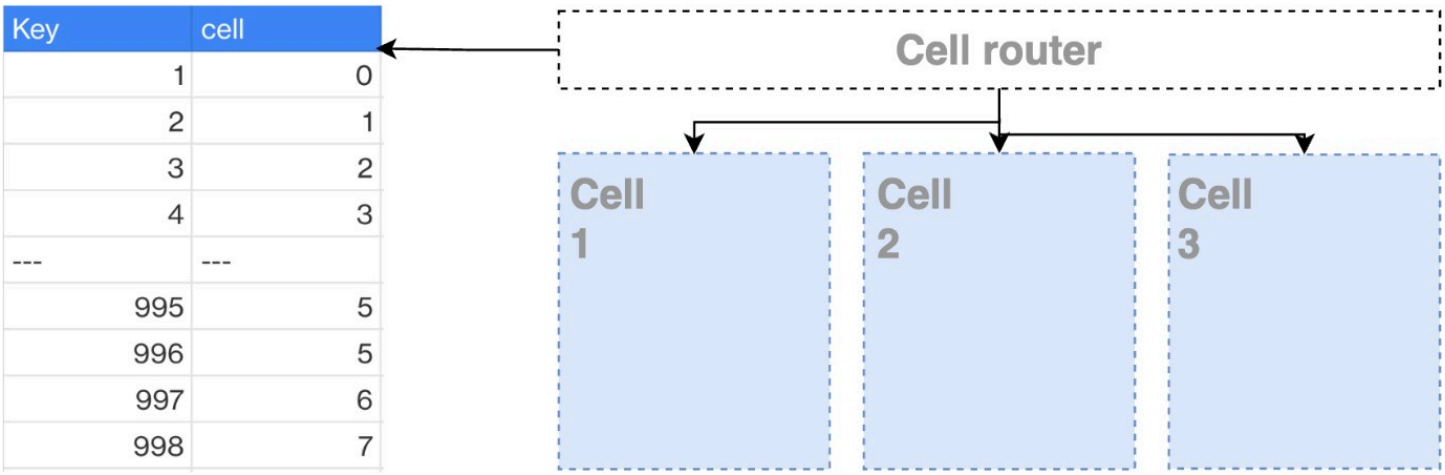
The following is a non-exhaustive list of partitioning algorithms presented without specific recommendations.

Topics

- Full mapping
- Prefix and range-based mapping
- Naïve modulo mapping or fixed partition number
- Consistent hashing
- A warning for all mapping approaches

Full mapping

A highly flexible yet expensive approach is to explicitly map every key to a cell. This comes with the downsides of a critical read and write dependency on the mapping table, a read-your-writes consistency requirement, and a large amount of state.



Full mapping

Advantages:

- Simple to implement.
- More control over distribution to control hot cells and to perform a cell migration.

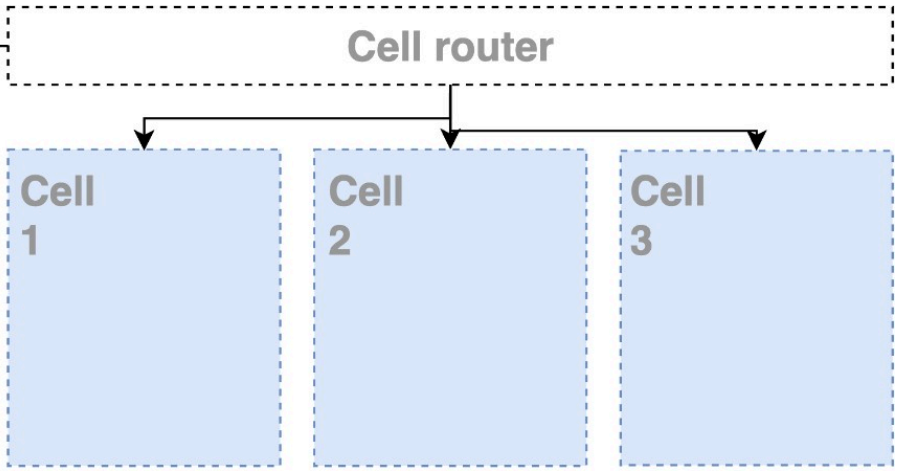
Disadvantages:

- Higher performance cost when cardinality is too high.
- If the map is kept in memory, it might have a longer cell router bootstrap time.

Prefix and range-based mapping

Prefix and range-based mapping map ranges of keys (or hashes of keys) to cells, and serves to offset the downsides of the full mapping approach while providing flexibility.

RangeStart	cell
0	0
1000	1
5000	2
10000	3
---	---
20000	5
30000	5
40000	6
50000	7



Prefix and range-based mapping

Depending on the granularity of your service, you can further reduce the cardinality by making ranges of groups of keys.

Ranges	cell
[0, 1000, 3000, 4000, 5000]	0
[38493, 18198, 93839, 83739, 8839]	1
[04949, 4273, 09584, 6839, 76272]	2
[938373, 374837, 7373, 9872, 2672]	3
---	---

Ranges as groups of keys

Advantages:

- Reduces the performance issue of full mapping, by grouping key and reducing the total cardinality.

Disadvantages:

- More likely to have a hot cell, as there is no control over which keys within each range might have the most traffic.

Naïve modulo mapping or fixed partition number

Naïve modulo mapping uses modular arithmetic to map keys to cells, typically on a cryptographic hash of the key. This scheme has an effective zero peak-to-average ratio (very even distribution) and requires minimal state (just the count of cells). But it suffers from high churn (cell reassignment) when adding or removing cells.

Advantages:

- Simple to implement.
- Avoids hot cells.

Disadvantages:

- Changing the number of cells requires the rebalance of all cells and their customers and tenants.

Consistent hashing

Consistent hashing is a family of algorithms that map keys to buckets (cells) with a small amount of fairly stable state and a minimal amount of churn when adding or removing buckets.

The Ring Consistent Hash (Karger, et. al.) is the consistent hashing scheme used in the Chord Distributed Hash Table, and is probably the best known. It can suffer from significant high peak-to-average ratios (uneven spread), although this can be offset with the tradeoff of additional state.

More recent algorithms improve upon the peak-to-average ratio and state requirements of the Ring Consistent Hash:

- A Fast, Minimal Memory, Consistent Hash Algorithm (Lamping and Veach)
- Multi-probe consistent hashing (Appleton and O'Reilly)

A general use of consistent hash in cell mapping is a system that is configured with a fixed large number (for example, tens of thousands) of logical buckets which are explicitly mapped to much smaller number of physical cells. Mapping a key to a cell is a two-step process. First, the key is mapped to its logical bucket using naïve module mapping. Then the cell for that bucket is located using a bucket to cell mapping table.

Key	Buckets	cell
82379017	0	0
92928392	1	1
2323922	2	2
8282838	3	3
	---	---
8832932	995	5
29382932	996	5
293292	997	6
2938292	998	7

Consistent hashing

Advantages:

- Changing the number of cells does not requires rebalancing all cells and their customers and tenants
- Simple to implement.

Disadvantages:

- Can suffer from significant high peak-to-average ratios (uneven spread).

A warning for all mapping approaches

Regardless of partition mapping approach, it's important to also use an override table to force specific keys to specific cells (except for full mapping, which natively provides this support). This can be useful for testing, quarantining, and special-case routing for particularly heavy partition keys.

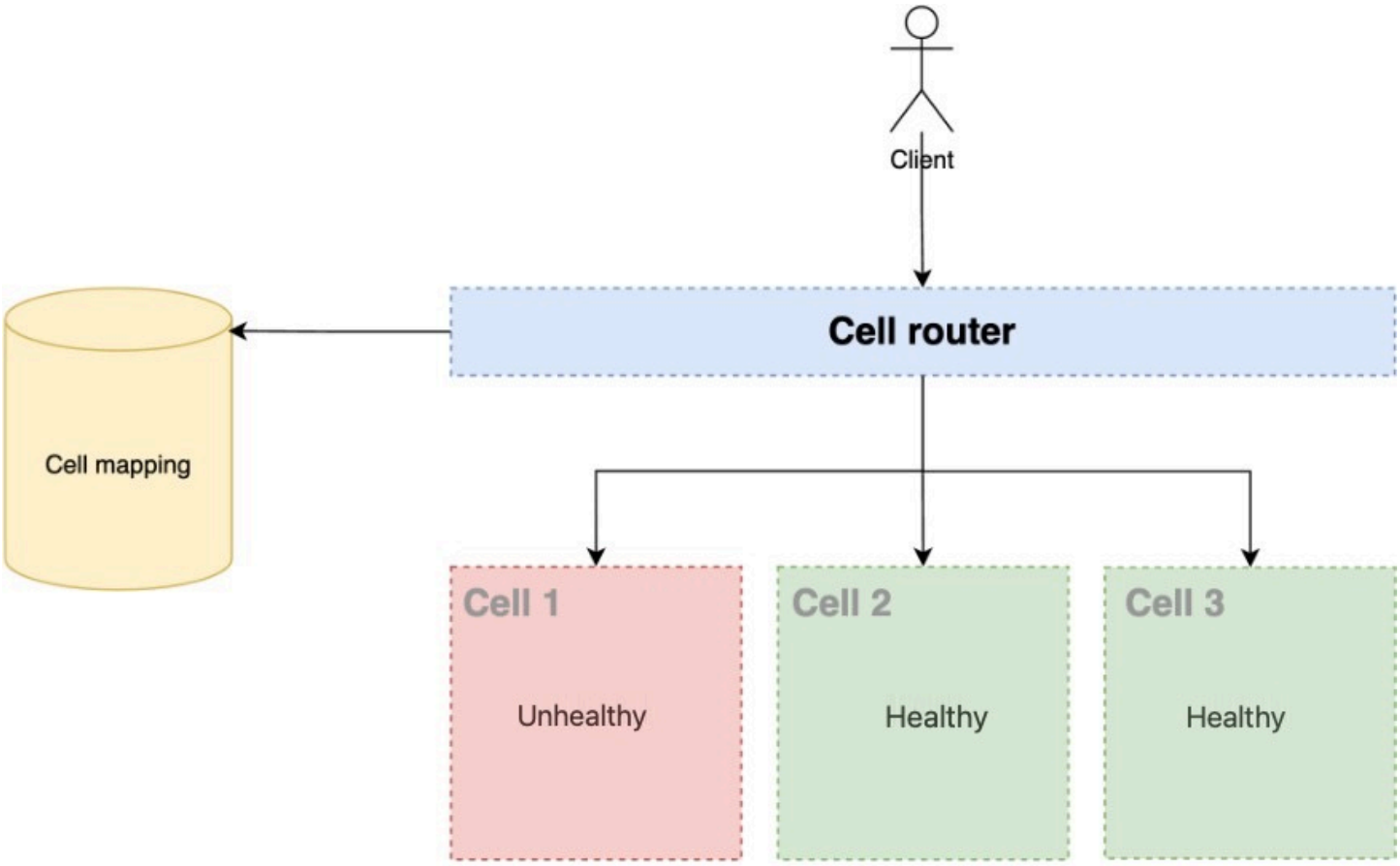
Another consideration is that the task of mapping a new customer to a cell and registering it in the cell router is the control plane's task. After this provisioning of the client and the cell router loads this configuration, the strategy defined in this section starts to work

Cell routing

The router layer is a shared component between cells, and therefore cannot follow the same compartmentalization strategy as with cells.

For the router layer, it's recommended that you distribute requests to individual cells using a partition mapping algorithm in a computationally efficient manner, such as combining cryptographic hash functions and modular arithmetic to map partition keys to cells.

To avoid multi-cell impacts, the routing layer must remain as simple and horizontally scalable as possible, which necessitates avoiding complex business logic within this layer. This has the added benefit of making it easy to understand its expected behavior at all times, allowing for thorough testability. As explained by Colm MacCárthaigh in Reliability, constant work, and a good cup of coffee, simple designs and constant work patterns produce reliable systems and reduce anti-fragility.



Avoiding multi-cell impacts

Cell router features to keep in mind:

- Be simple as possible, but not simpler.
- Have request dispatching isolation between cells.
- Minimize the amount of business logic in this layer.
- Abstract underlying cellular implementation and complexity from clients.
- Fast and reliable.
- Continue operating normally in other cells even when one cell is unreachable.

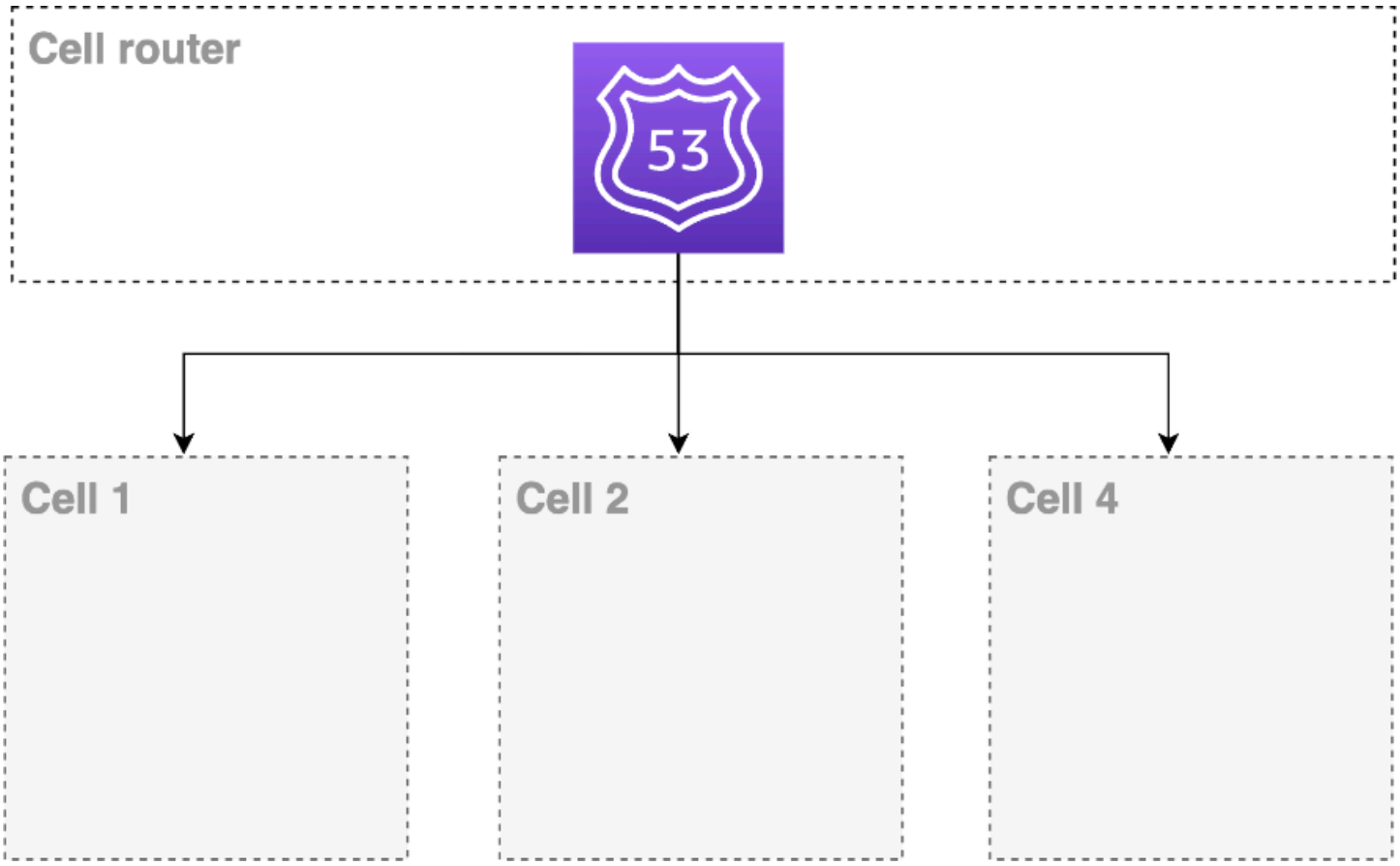
Although the previous diagram symbolizes the cell router seeking information, this is not necessarily the only approach that can be taken. There are several options for designing a cell router with their advantages and disadvantages. The objective is that the cell router has the mapping state of each partition key for its respective cell. The following is a non-exhaustive list of design that can be applied to design a cell router.

Topics

- Using Amazon Route 53
- Using Amazon API Gateway as cell router
- Using a compute layer like Amazon EC2, Amazon ECS or Amazon EKS and Amazon S3 for cell mapping as cell router
- Routing no-HTTP requests
- About resilience of the cell router

Using Amazon Route 53

Amazon Route 53 is a highly available and scalable Domain Name System (DNS) web service. You can use Route 53 to perform the functions of cell router in your architecture, doing DNS routing, and health checking. Route 53 is a service that offers a SLA of 100% for the data plane and can be used to redirect the traffic to correct cell, or even failover in case of an impairment at an Availability Zone or Region. Give each tenant a custom DNS record they use to reach your service, then configure the DNS record to point at a specific cell to which the tenant has been assigned.

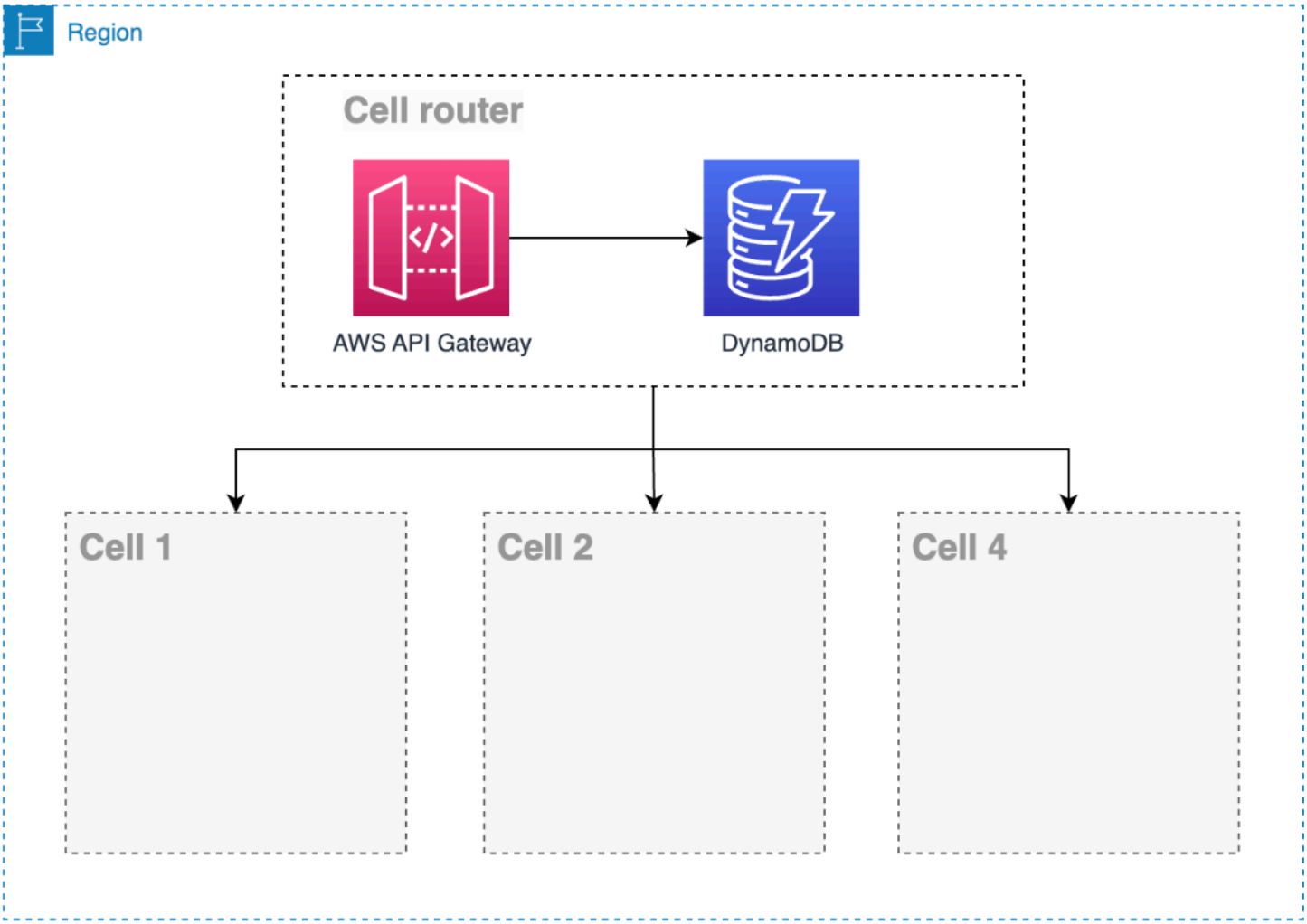


Using Amazon Route 53

Amazon Route 53 can also be combined with Route 53 application recovery. The Application Recovery Controller also helps you manage and coordinate recovery for your applications across AWS Availability Zones (AZs) or Regions. The features made available by it can help in case of unavailability of an AZ, Region, or even in case of gray failures, where an evacuation of the AZ is a better alternative until the problem is found.

Using Amazon API Gateway as cell router

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale. It's a serverless regional service and has a good fit to be your cell router. It has native integration with many AWS services, caching support, throttling, rate limit, canary deployments, usage plan configuration and many other features. DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability com latency single digit in milliseconds and with an SLA of 99.99% and 99.999% with global tables enabled.



Using Amazon API Gateway

DynamoDB also features the Amazon DynamoDB Accelerator (DAX) is a fully managed, highly available in-memory cache for Amazon DynamoDB that delivers up to a 10X performance improvement from milliseconds to microseconds, even at millions of requests per second.

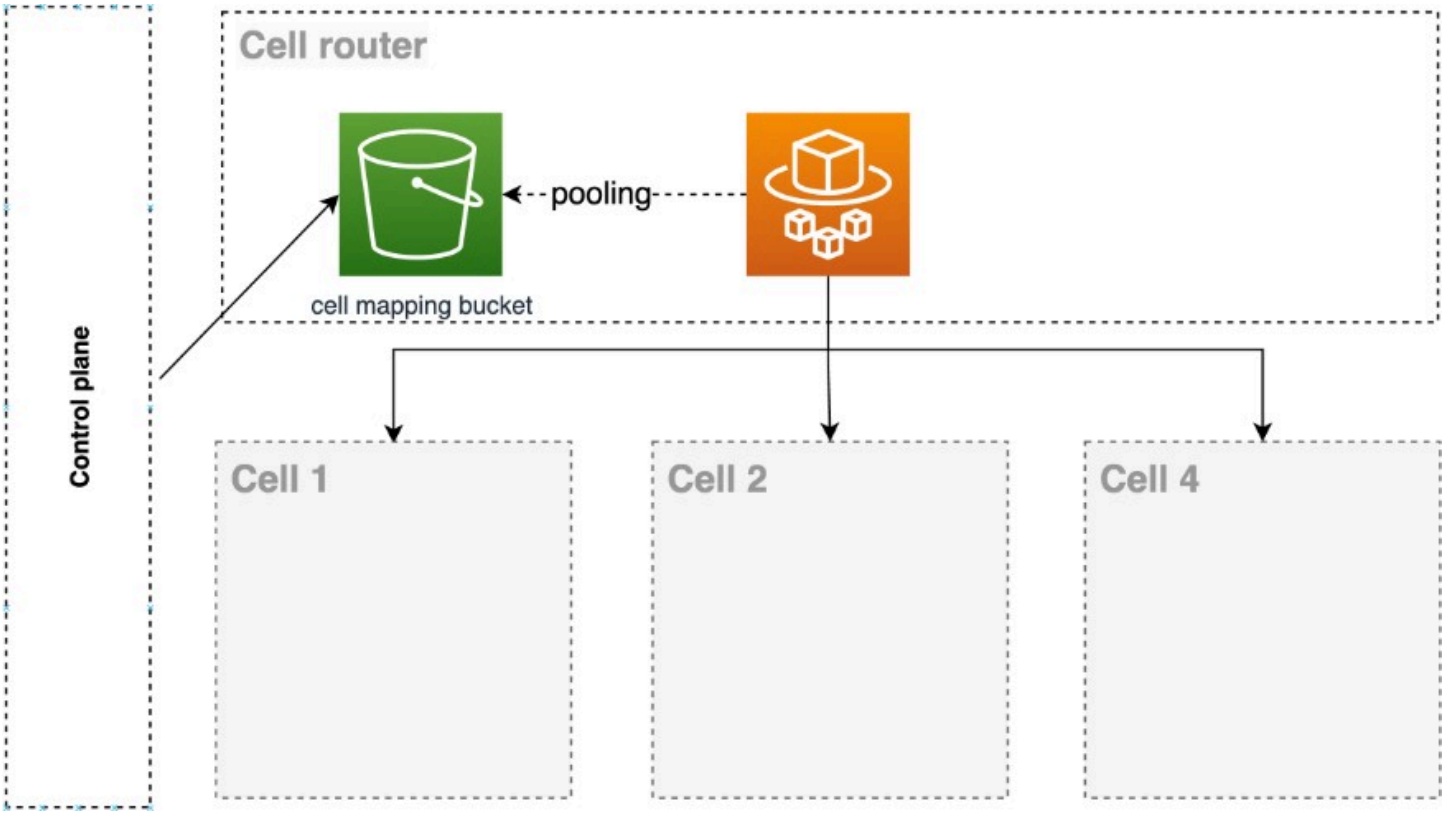
DynamoDB is a powerful serverless and regional NoSQL database, but according to your workload needs, any database can be used in this model of cell router architecture.

Using a compute layer like Amazon EC2, Amazon ECS or Amazon EKS and Amazon S3 for cell mapping as cell router

Another approach to designing your cell router is to have the control plane write the cell mapping to an S3 bucket and a computer layer, whatever it is, it could be an EC2 instance, an Amazon ECS or Amazon EKS cluster, or AWS Lambda, whichever fits best. It is worth emphasizing that at this

cellular router layer, the only responsibility should be to inspect the request data and identify which cell the request should be forwarded to.

The complexity of the access pattern, the cardinality of your partition key, the number of cells, all these factors can influence what is the best approach to keep your cell router up to date with cell mapping. Avoiding overload in distributed systems by putting the smaller service in control is a good article with some more alternatives on how we do this synchronization between data plane and control plane in AWS. It can be a basis for the synchronization process of your cell router.



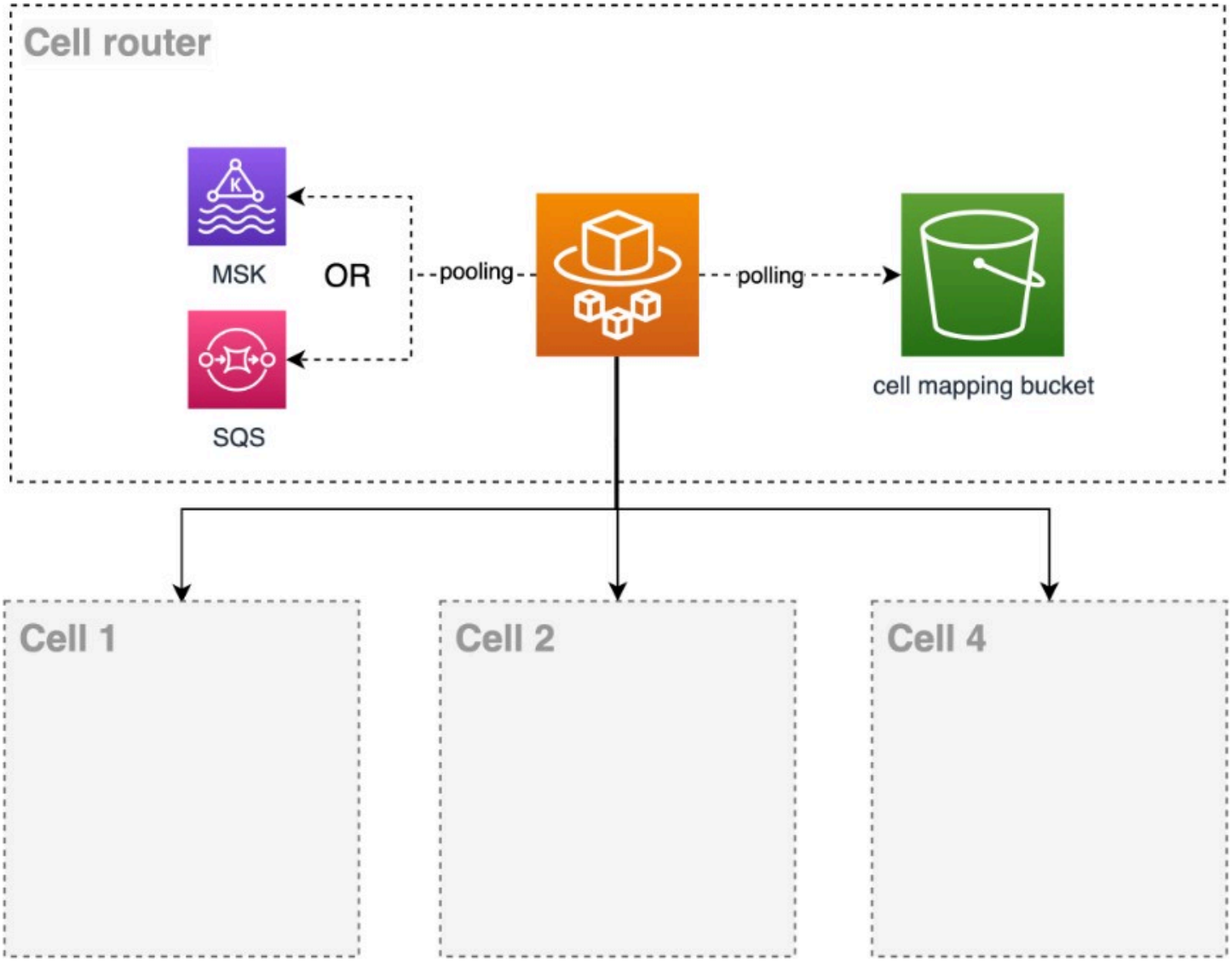
Using a compute layer

In this example, the cell mapping lives in memory on the router. With each change in the S3 bucket, another process or thread is in listener mode and updates the memory map when necessary.

Routing no-HTTP requests

According to each business, we can have APIs of different types as an input interface for your workload. The examples so far have been routers based on HTTP requests. But nothing limits that your cellular router cannot be an event or messaging broker. You can have a payments API, where you offer your customers an asynchronous API using Amazon SQS queues. Where you have a queue

to request a payment and a queue to process the response to that payment. In this example, the cell router consumes the payment request topic and delivers it to the correct cell to carry out the transaction.



Routing no-HTTP requests

In this example the entry point is an Events/Messages API which could be an Amazon SQS or an Amazon MSK. As in the previous example, cell mapping lives in memory in the router. With each change in the S3 bucket, another process or thread is in listener mode and updates the memory map when necessary.

About resilience of the cell router

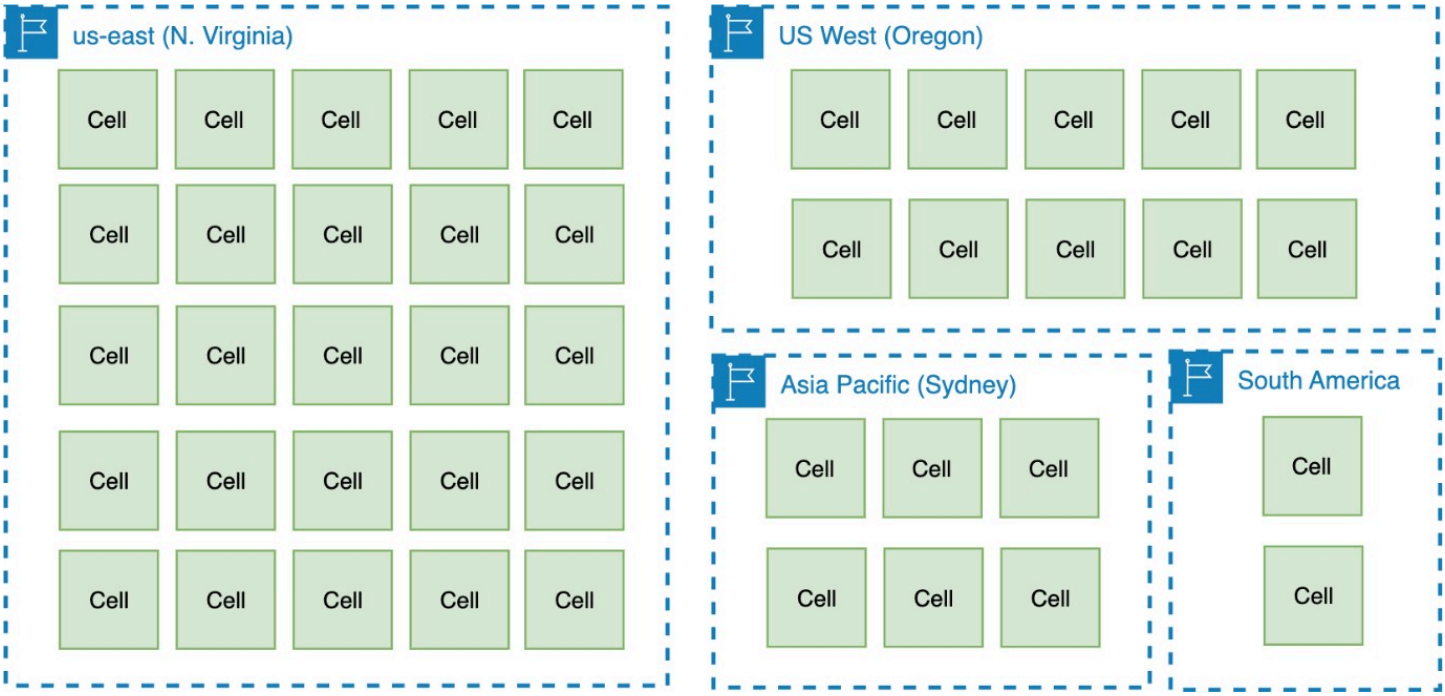
In a cell-based architecture, the only component that has the shared state of all cells is the cell router. It presents itself as a single point of failure. Therefore, it is essential that it be built of with maximum reliability and also as a cellular component, regardless of whether your cell strategy is AZ independent or non-AZ independent, all the recommendations described so far and later must also be followed for the cell router. Mainly issues of service limits, size and observability.

In other words, the routing layer still has to scale *infinitely*, but the set of problems that you have to solve for scaling the thinnest possible layer should be a subset of the scaling challenges that non-cellularized application would have to face.

Cell sizing

Cell-based architectures benefit from capping the maximum size of a cell, and using consistent cell sizes across different installations (for example, AZ or Regions). When thinking about small or large cells there are three opposing forces on choosing a cell size:

- Big enough to fit the largest workloads.
- Small enough to test at full scale (and to operate efficiently) that is equal lower risk of scaling cliffs, below the AWS account limits, etc.
- Big enough to gain economies of scale benefits.



Cell sizing

The maximum cell size will vary per-service. The optimum point will depend on the service and customer behavior, but needn't be extremely large for any service. There are tradeoffs to be considered in how to select the maximum size of a cell:

Smaller cells	Larger cells
Will have more cells – By having smaller cells, there is a need to deploy more cells to operate in general. The smaller the cells, the more replicas of your workload you will have to manage.	**Will have fewer cells –** By having larger cells, you need to deploy fewer cells to operate. The larger the cells, the less replicas of your application you need to manage
Cell outage or drain impacts small percentage of compute fleet – As in the example, with a smaller number of cells. any event that affects them, will also be affecting a smaller portion of their infrastructure	**Cell outage or drain impacts large percentage of compute fleet –** As an effect of having large cells, any event affects a larger portion of your infrastructure
Less likely to hit scaling limitation – In AWS all resources have limits and quotas by Region	**More likely to reach scaling limitation –** Larger cells will use more computational

Smaller cells	Larger cells
and account. With smaller cells, the probability of a single cell reaching these limits is reduced. There are often unseen/unknown limitations in implementations that manifest themselves at size and scale and with cells, these also have their impact reduced.	power, being more likely to reach Region and account limits.
Reduced scope of impact – If with 10 cells, each one has 10% with their customers, with 100 each one has 1%. Naturally when a cell fails, the scope of impact will be smaller.	**Reduced splits –** According to your partition key, isolating client workloads to individual cells rather than having to split individual client workloads across cells.
Easier to test – As a good practice, cells should have stipulated limits and quotas and tests should be implemented to test these limits. With smaller cells it is easier and even cheaper to test the cells.	**Easier to operate –** Considering that each cell is a complete replica of its workload, Operate 5 for example is easier than operating 10, 20 or 30 workloads. Even so, it is important to build the necessary tools to automate the operation of cells, even environments with "large" cells, which can grow to tens, hundreds or more cells.
Less idle resources – Smaller cells have less computational capacity, so the probability of having idle resources is lower.	**Better capacity utilization –** Larger cells will support more clients and more traffic, thus having greater economy of scale in resources.

The benefit of a cell having high scalability or being treated as a scale unit, comes from the ability to have its maximum limit known as recommended in REL01-BP01 Aware of service quotas and constraints, that is, according to your business:

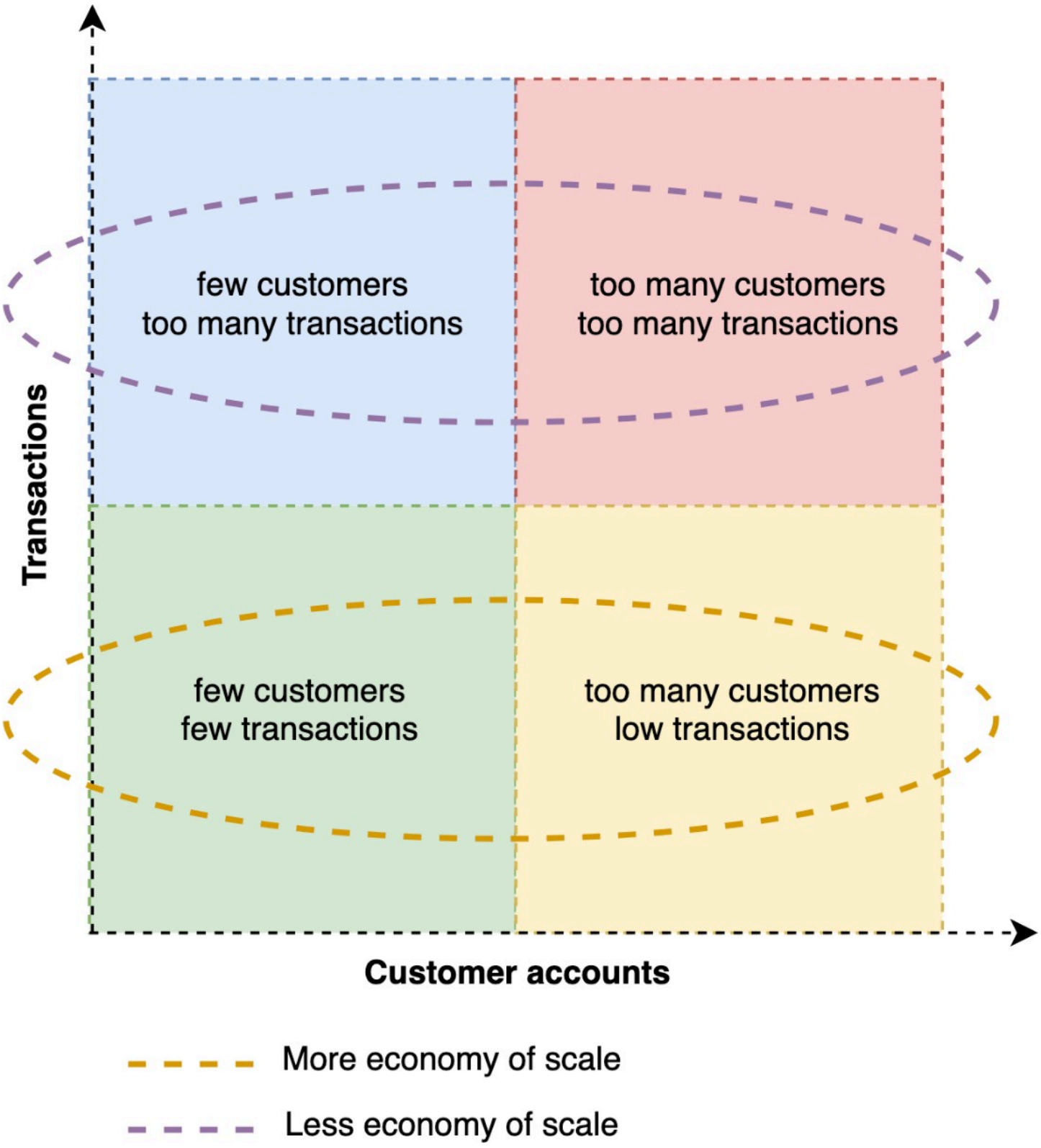
- How many transactions per second can a cell handle?
- How many customers or tenants does it support?
- How many GB of transfer per second or stored capacity does it support?

Define your scaling dimensions

This subject of cell sizing is closely related to your partition key, it is what will define in which cell the workload traffic will be directed and stored. In cell sizing, we are focusing on *how much* of this factor a cell will support before it needs to scale. Remember that in this case, your architecture does scale-out, it scales by adding more cells that have the same limit capacity.

It's good to keep in mind the most granular and independent unit in your system. The most obvious choice might be the `client ID`. But let's say, for example, that you define that a cell in your workload can handle 10K TPS, and a single client of your workload starts to grow beyond that number. In this scenario, your system becomes unable to scale-out, being forced to scale-up (if possible) or simply making the system unable to serve this large customer.

Defining more than one scale unit dimension will help you handle clients that are much larger than most, true outliers, having a dedicated cell for them or even more than one. However, the latter can still cause other problems, such as the need to have a scather/gather router. *Dedicated cells* are important for the enterprise because the architecture opens up a market for dedicated single tenancy. If a customer really wants it, and is willing to pay for it (and a surprising number are), you can dedicate a cell totally to them. This could be a lot of additional revenue and also could make customers happier, and safer.



Scaling dimensions

One last factor that cannot be overlooked is cost. When defining the size of your cells, also calculate how much each cell will cost you. This calculation can help you decide how multi-tenant your system is, which can increase or decrease the economies of scale and margin advantage that your business might have.

Know and respect your cell's limit

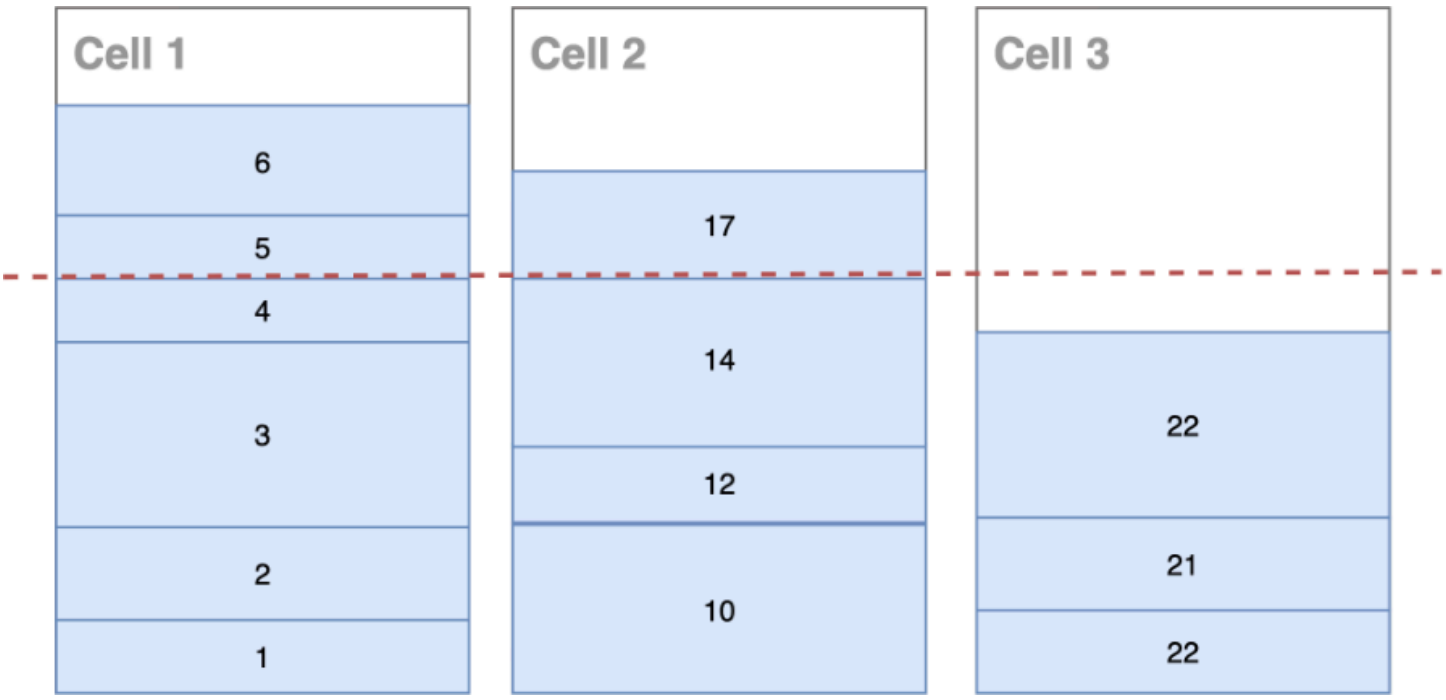
Cell sizing is fully related with the limits of traffic that your cell can support without impact negatively the cell's customers. Using load shedding to avoid overload is fundamental and can reduce unknown behaviors in your workload, once you know how much traffic your cells can handle. You can determine this through load testing and chaos engineering. Some AWS services can help to do this control, such as Amazon API Gateway, which supports rate limiting at the API level (by resource and method) and at stage-level.

If you are using a more customized approach for your cell router, you can implement algorithms like token bucket. An example of the use of token bucket is made by Amazon EC2 API.

Cell placement

Cell placement it is another responsibility of the control plane and refers to the activities of onboarding new tenants and customers and also creating the cells themselves. To do these activities some level of observability is required such as:

- Capacity of each cell
- Used capacity of each cell
- Percentage of usage of each tenant or customer
- Quotas and limits of each cell in their respective AWS account



Cell placement

In an ideal world, it is important to find the percentage of utilization and the limit that each of your cells will work with the greatest possible predictability and stability. It is not because your cell supports 10K TPS that you must constantly be operating on the threshold of this limit, as already recommended in REL01-BP06. Ensure that a sufficient gap exists between the current quotas and the maximum usage to accommodate failover.

For data and state heavy workloads, the problem of allocation becomes a core competency. This is classically a pretty deep area of allocation, scheduling, and forecasting, that is, it is your strategy to evenly distribute your traffic among the cells. For example, when a customer or partitioned key starts to dominate a cell, you need to migrate some tenants or customers to other cells. In addition to the entry of new tenants or clients, there are other scenarios where a placement strategy is necessary:

- The dimensions of each cell (might change over time, or might be fixed).
- The dimensions of each partition key (changes over time).
- The cost of moving a partition key between cells.
- The benefit of co-tenanting certain partition keys or having affinity.

Cell migration

Cells should not share state or components between cells. Depending on the workload, a cell migration strategy might be needed to migrate data/customers from one cell to another. A possible scenario when a cell migration may be needed is if a particular customer or resource in your workload becomes too big and requires it to have a dedicated cell.

Stateful cell-based architectures will almost certainly require online cell migration to adjust placement when cells are added or removed. One consideration of online cell migration is handling mapping decisions during the transitionary period. This may involve cross-cell redirects, performing multiple iterations of the mapping algorithm when necessary, or both, against different versions of the mapping algorithm state.

Another consideration is how to safely migrate the state. This will be system-dependent but at a high-level will likely consist of the following phases:

- **Clone** the data from the current location into the new location, as a non-authoritative copy.
- **Flip** the new location copy to be *authoritative*.
- **Redirect** from old location to new location.
- **Forget** the data from the old location.

Another approach is to use careful coordination between the router and the cells, for example using the control plane to migrate clients from one cell to another and ensuring this state transition before the cell is ready to receive traffic. In this case, dependencies between cells are avoided or kept to a minimum, as these dependencies have been influenced across cells and therefore decrease fault isolation.

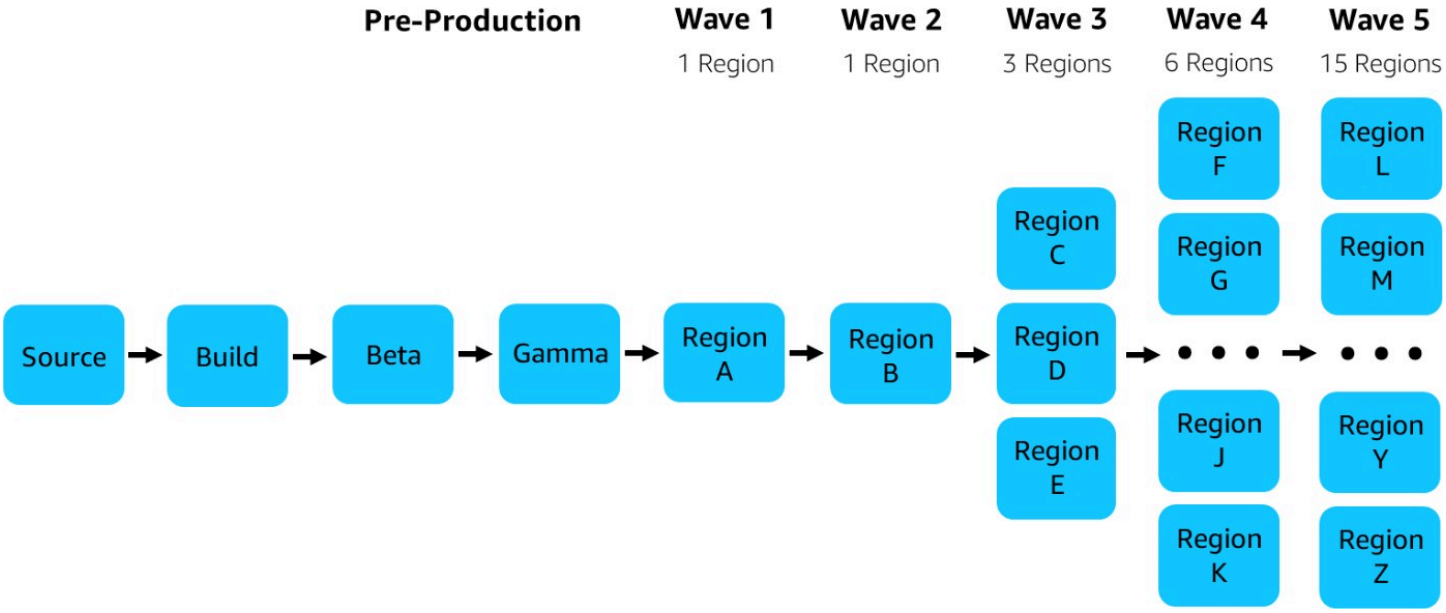
Cell deployment

Unless you are already working on a workload that is multi-Region, you currently have one instance of your workload to develop, deploy and operate. Now with your workload using a cell-based architecture, you have tens, hundreds or even thousands of instances of your workload to deploy and operate, depending on the limits, scale units, and cell size you set. In summary, it is a very complex challenge to deploy in a production environment.

Cell-based architecture brings a new dimension to its context, which is not trivial for development. If today you have to deploy your source code in an environment (development, pre-production, and

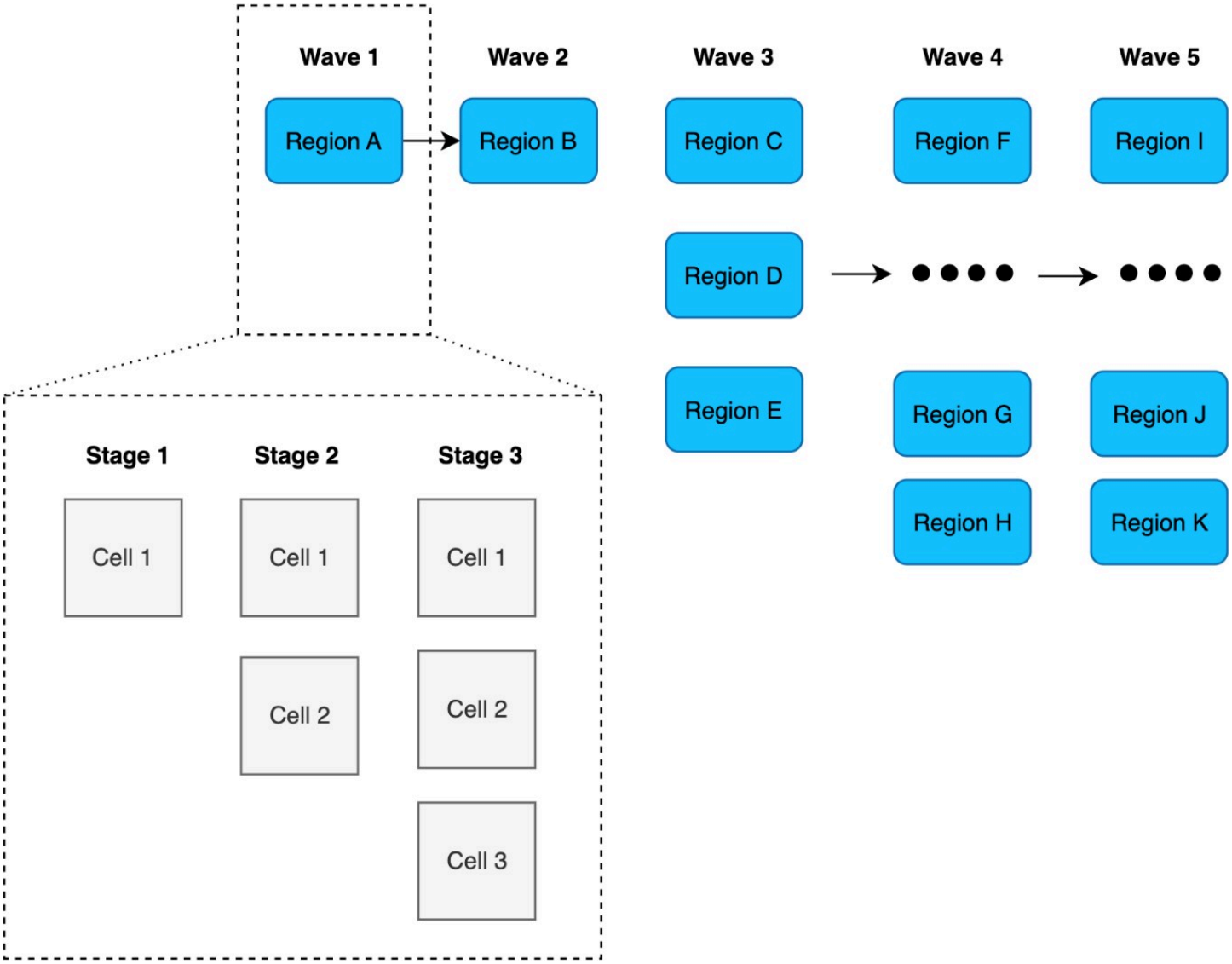
production) in an Availability Zone or Region, now you have to deploy a cell on all these aspects as well. To avoid problems, it is essential to have an automated CI/CD pipeline from the beginning. At Amazon, we have a strong culture of continuous delivery, which you can find out more about in My CI/CD pipeline is my release captain.

In the following diagram, we have an Amazon service pipeline type, where each service is deployed in phases until it reaches general availability for all Regions



Amazon service deployment

This is a great way to reduce the impacts of infrastructure failures, bugs, and other errors that can impact customers. With cell-based architecture, these are also deployed in phases, as in the example in the following diagram:



Cell deployment

The benefits of fault isolation and blast radius reduction with cell-based architecture are not only when processing customer traffic, but also when deploying new features and fixing bugs. With your customers or in the partitioning model you chose, your deployment model will also follow this same concept, deploying one or more cells at a time, and when identifying any sign of failure, you can rollback, thus reducing the number of clients that were exposed to this failure.

The previous example was given based on the Amazon deployment model, but the important point here is to deploy in waves, cell by cell or set of cells. It doesn't matter if the cell strategy you chose is non-AZ independency or AZ independency. To delve into other important issues, a good starting point is the Reliability and Operational Excellence best practices of the Well-Architected Framework:

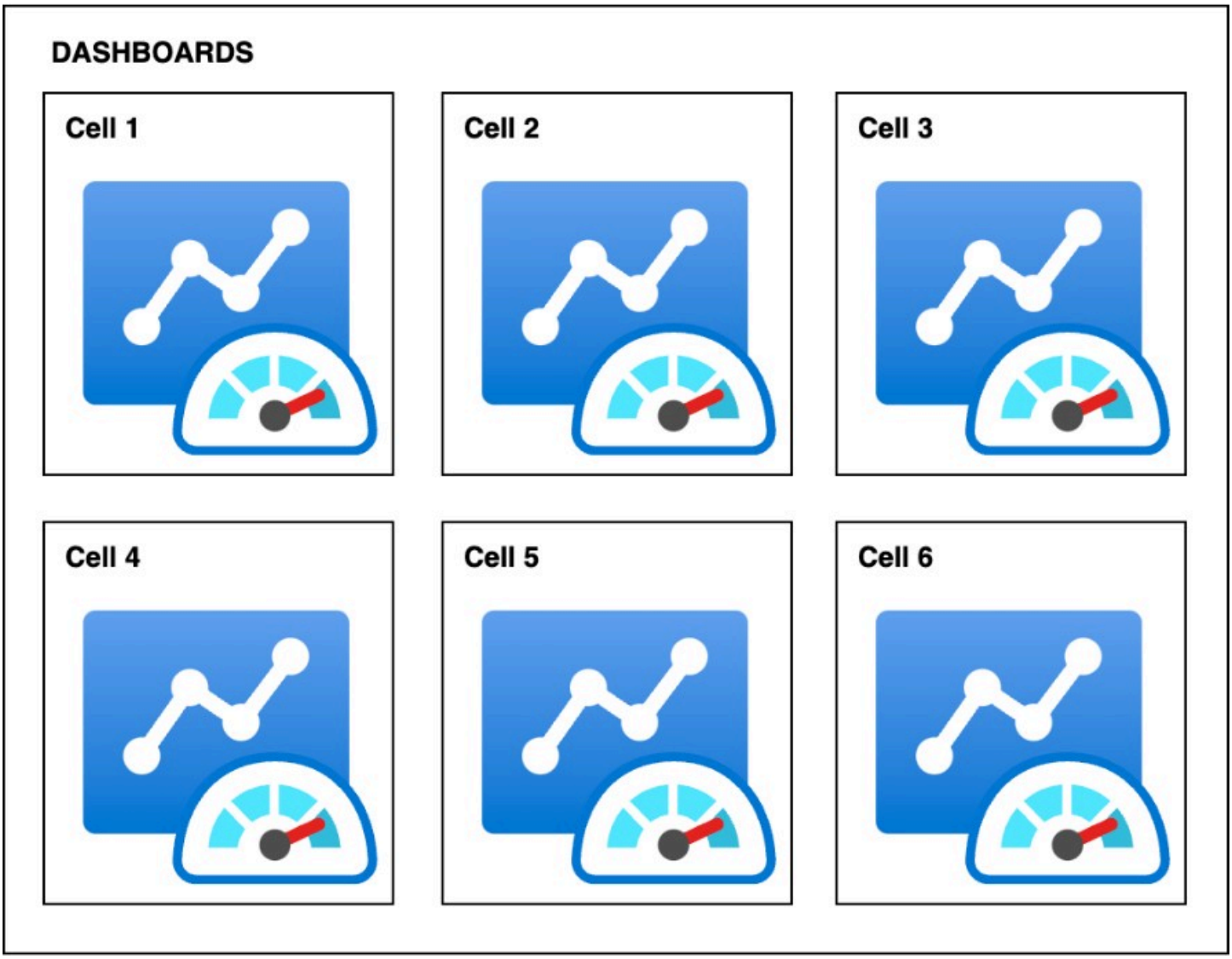
- REL08-BP05 Deploy changes with automation
- OPS05-BP10 Fully automate integration and deployment
- OPS06-BP01 Plan for unsuccessful changes
- OPS06-BP07 Fully automate integration and deployment
- OPS06-BP08 Automate testing and rollback

AWS services that can help you with this implementation are:

- AWS CodeCommit
- AWS CodeBuild
- AWS CodePipeline.

Cell observability

Doing cell-based requires a lot of automation and a set of very specific tools. As the composition of a cell itself can vary from business to business, many things will have to be built around your business. Observability is one of them. If you had a stack before, now you have many stacks, and to take advantage of all the benefits of a cell-based architecture it is necessary to have some ideas in mind.



Dashboards to observe cells

Your entire observability stack needs to be cell-aware. Best practices like How do you design telemetry now need to give you a view of each cell in its individuality. It is important to be able to track each request and identify which cell it is destined for. For more information on how we do this on Amazon (though, generally not cell-based) see:

- Amazon Builders' Library: Instrumenting distributed systems for operational visibility
- Amazon Builders' Library: Building dashboards for operational visibility

There is also a great hands-on experience lab for you on the wide variety of tools that AWS offers to set up monitoring and observability in general on your applications. The main point here is

that you do this on a cell-by-cell level, so that you have a new dimension to observe and react to accordingly.

Best practices

Your current instance/stack is your cell zero

When starting to plan your migration to a cell-based architecture, consider your current stack as cell zero. Adding the router layer above and little by little distributing your workload traffic according to your cell and cell partition strategy.

Start with multiple cells from day one

Since the cell is a replica of your workload to handle a portion of your customers, keep in mind from day 1 to have more than one cell. This will bring you the adverse issues and experience needed to operate in this type of environment, reducing surprises.

Start with a cell migration mechanism from day one

Migrating clients from one cell to another is a tricky topic, depending on the nature of your workload it may require a lot of coordination and orchestration. Have that in mind from day one. So even if a customer quickly becomes too big for a cell, or the cell size you initially set isn't ideal, you have the mechanism to move your customers to other cells.

Perform a failure mode analysis of your cell

Since the cell is a failure isolation boundary, it is a good exercise to analyze which services make up your cell and what is the effect of each component failing partially or completely and make sure that other cells are not impacted.

A worksheet containing the component and cause, probability, mitigation of the failure will already give you good ideas of what can happen and how to react accordingly.

Conclusion

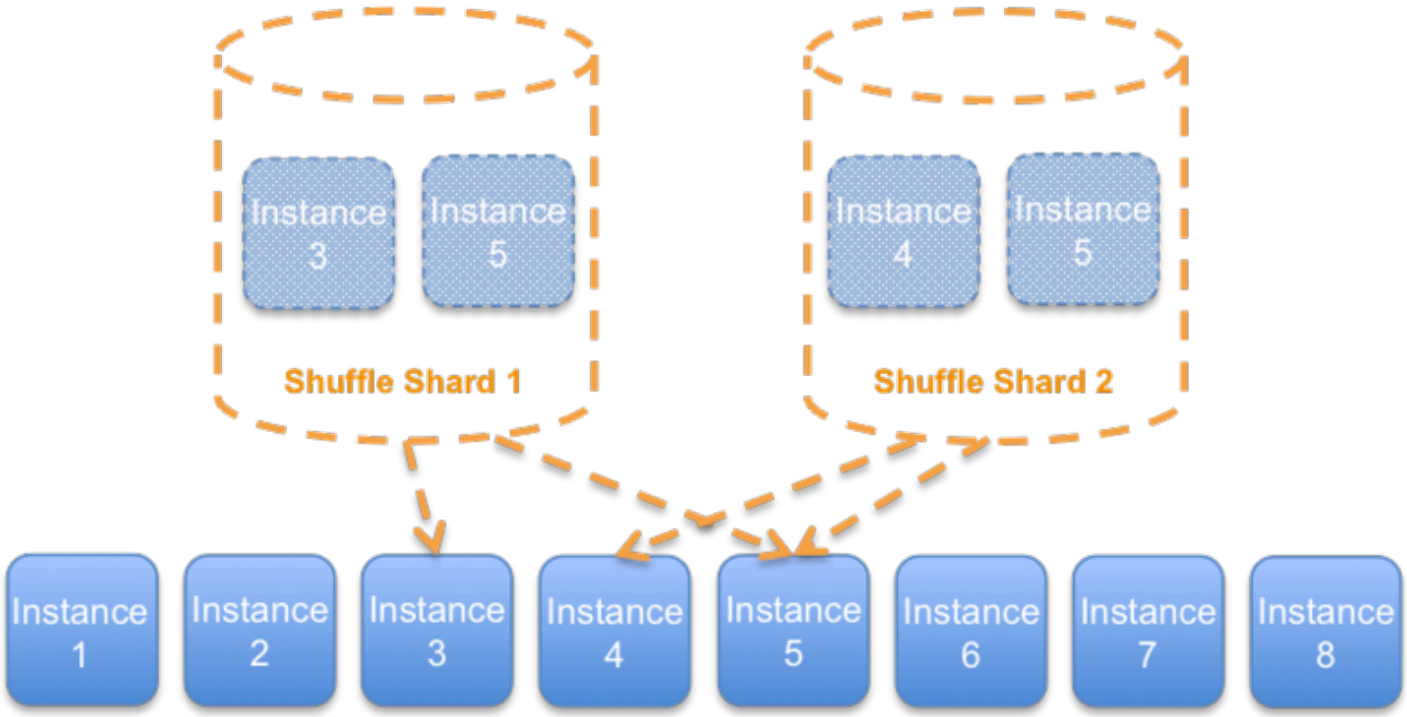
Cell-based architecture can bring a higher level of isolation, predictability, and testability to your workload. It is extremely important to understand the trade-offs of using this architectural model and that not all your business workloads are going to require extreme levels of resiliency.

This guidance focused on workload architecture of cell-based architecture, but one aspect that cannot be overlooked is operational excellence, with cells you now have dozens if not hundreds of replicas of your workload to operate and evolve. All of the operational excellence best practices recommended by the Well-architected framework should have their attention reinforced on workloads that use this architectural style.

FAQ

What about shuffle-sharding?

A question often asked is whether shuffle-sharding is the same as cell-based or how they are related. Although shuffle-sharding is an excellent fault-isolating mechanism, they are not the same thing. The basic idea of shuffle-sharding is to generate shards as we might deal hands from a deck of cards. Take the eight instances example. Previously, we divided it into four shards of two instances. With shuffle-sharding, the shards contain two random instances, and the shards, just like our hands of cards, might have some overlap.



Shuffle-sharding

In a cell-based architecture, a cell should be self-contained, not share its state. We can use shuffle-sharding within a cell, but cross-cells should not be used by definition. Shuffle-sharding can also be a bit trickier for stateful components. To learn more about shuffle-sharding, here are two great articles:

- Workload isolation using shuffle-sharding
- Shuffle Sharding: Massive and Magical Fault Isolation

Contributors

Contributors to this document include:

- Robisson Oliveira, Sr. Cloud Application Architect, Amazon Web Services

Further reading

For additional information, refer to:

- *AWS Well-Architected Framework Reliability Pillar* whitepaper, specifically REL10-BP04 Use bulkhead architectures to limit scope of impact.
- AWS Architecture Center
- Shared Responsibility Model for Resiliency in the *Disaster Recovery of Workloads on AWS* whitepaper.
- AWS Fault Isolation Boundaries whitepaper
- Millions of Tiny Databases

Videos

- re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338)
- Physalia: Cell-based Architecture to Provide Higher Availability on Amazon EBS
- re:Invent 2022 - Camada Zero: A real-world architecture framework (PRT268)

Document history

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Initial publication	Whitepaper first published.	September 20, 2023

Note

To subscribe to RSS updates, you must have an RSS plug-in enabled for the browser that you are using.

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.