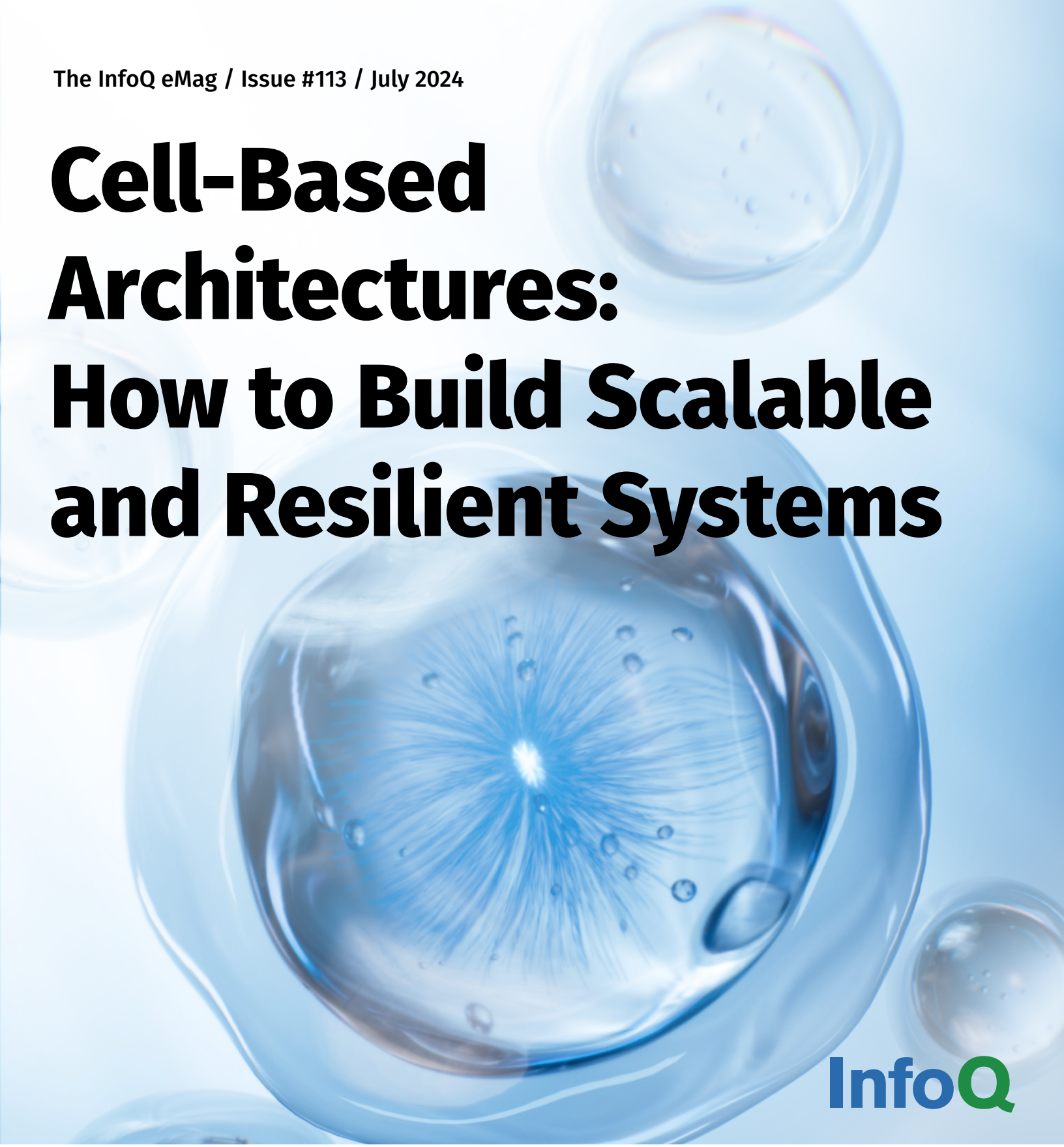
The InfoQ eMag / Issue #113 / July 2024

Cell-Based Architectures: How to Build Scalable and Resilient Systems



InfoQ

How Cell-Based Architecture Enhances Modern Distributed Systems

Taking Advantage of Cell-Based Architectures to Build Resilient and Fault-Tolerant Systems

Architecting for High Availability in the Cloud with Cellular Architecture

Cell-Based Architectures: How to Build Scalable and Resilient Systems

IN THIS ISSUE

PRODUCTION EDITORS Anamaria Bota & Roxana Bacila / COPY EDITORS Maureen Spencer & Susan Conant
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

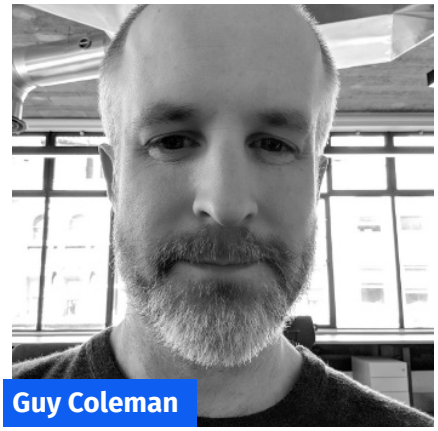
CONTRIBUTORS

Yury Niño Roa

Cloud Infrastructure Engineer at Google with 8+ years of experience designing, implementing and managing the development of software applications using agile methodologies such as scrum and kanban. 3+ years of DevOps and SRE experience supporting, automating and optimizing mission-critical deployments, leveraging configuration management, CI/CD, and DevOps processes. Professor of Software Engineering and Researcher.

Erica Pisani

is currently a Sr. Software Engineer on the integrations team at Netlify. She's worked in a number of startups across a variety of industries throughout her career, including financial/small business technology, human resources software, and pharmaceutical research technology.



Guy Coleman

A tech lead of modern software teams with over 20 years of experience across gaming, mobile, finance and plenty of things in between. Spends most of his time with Java cloud-native architectures nowadays, with an emphasis on performance, observability and cloud platforms.

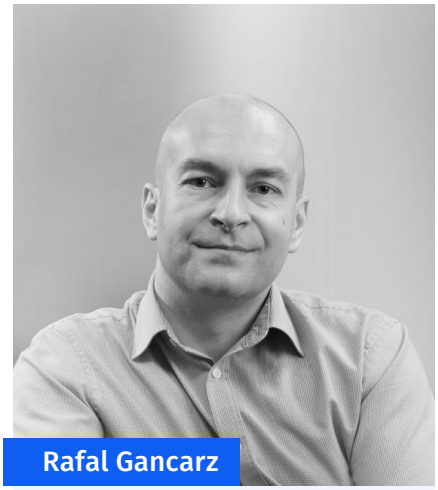
Stefania Chaplin

Stefania's (aka DevStefOps) experience as a Solutions Architect within DevSecOps, Security Awareness and Software Supply Chain Management means she's helped countless organizations understand and implement security throughout their SDLC. As a python developer at heart, Stefania enjoys optimizing and improving operational efficiency by scripting & automating processes and creating integrations.

Chris Price

is a Software Engineer at Momento, where he helped design and build Momento's cellular architecture and the corresponding CI/CD automation. His interest in infrastructure automation began at Puppet Labs, during the genesis of the DevOps movement. After 5 years in a tech lead role at Puppet, Chris moved on to AWS, where he helped build the foundation for the brand new AWS MediaTailor service, a core component of Amazon's NFL Thursday Night Football broadcasts and one of the first services inside of AWS to launch with a cellular architecture on Day 1.

A LETTER FROM THE EDITOR



Rafal Gancarz

Rafal is an experienced technology leader and expert. He's currently helping Starbucks make its Commerce Platform scalable, resilient and cost-effective. Previously, Rafal has been involved in designing and building large-scale, distributed and cloud-based systems for Cisco, Accenture, Capita, ICE, Callsign and others. His interests span architecture & design, continuous delivery, observability and operability, as well as sociotechnical and organisational aspects of software delivery.

The IT industry has been grappling with mastering distributed systems for many decades. As these systems become increasingly complex, they continue to present considerable challenges to organizations developing digital products. Arguably, one of the most challenging aspects of distributed systems is reliability in the face of failure, particularly as modern distributed systems utilize large numbers of physical and virtual resources, including networking, computing, and storage.

From the early days of the low-level network protocols and the web, distributed systems have undergone a significant transformation, evolving into Service-Oriented Architectures (SOA) and, more recently, microservices. With cloud computing accelerating the growth of distributed systems even more rapidly, the distribution level has increased significantly. Function-as-a-Service (FaaS) and edge computing push the boundaries for the number of computing and processing agents into millions.

As L Peter Deutsch and others aptly point out in The Eight Fallacies of Distributed Computing, it's a fallacy to assume that networks are always reliable. Even the most basic cloud services heavily rely on networking. This underscores the ongoing relevance of Deutsch's insights, which were first shared in the nineties. In the era of cloud and edge computing, these observations are not only still relevant but perhaps even more so.

Cell-based architecture has emerged as a response to many challenges associated with distributed systems. First and foremost, it employs a bulkhead pattern to isolate failures to a fraction of the affected infrastructure footprint and prevent widespread impact. But that's not all it can offer. Cells can also help organize large architectures into domain-bound deployment and delivery units, which provides essential sociotechnical benefits.

In this "Cell-Based Architecture" eMag, we aim to present many benefits this pattern offers to

modern distributed architectures. The eMag articles, written by industry experts, will take readers on a journey of discovery and provide a comprehensive overview and in-depth analysis of many key aspects of cell-based architectures, as well as practical advice for applying this approach to existing and new architectures.

We start with “How Cell-Based Architecture Enhances Modern Distributed Systems,” where Erica Pisani and I provide an essential introduction to cell-based architecture (CBA). The article covers the origin of this approach, describes key concepts, and discusses many important considerations relevant to a successful implementation.

With a solid understanding of cell-based architecture at your fingertips, we move to “Taking Advantage of Cell-Based Architectures to Build Resilient and Fault-Tolerant Systems” by Yury Niño Roa. Here, the author focuses on cell-based architecture’s resiliency and fault-tolerance benefits through the lens of observability. Roa argues that comprehensive observability, already quite crucial for microservice-based architectures, is paramount to successfully implementing CBA, given the additional complexity of routing traffic to correct cells and the need to monitor cells’ health constantly.

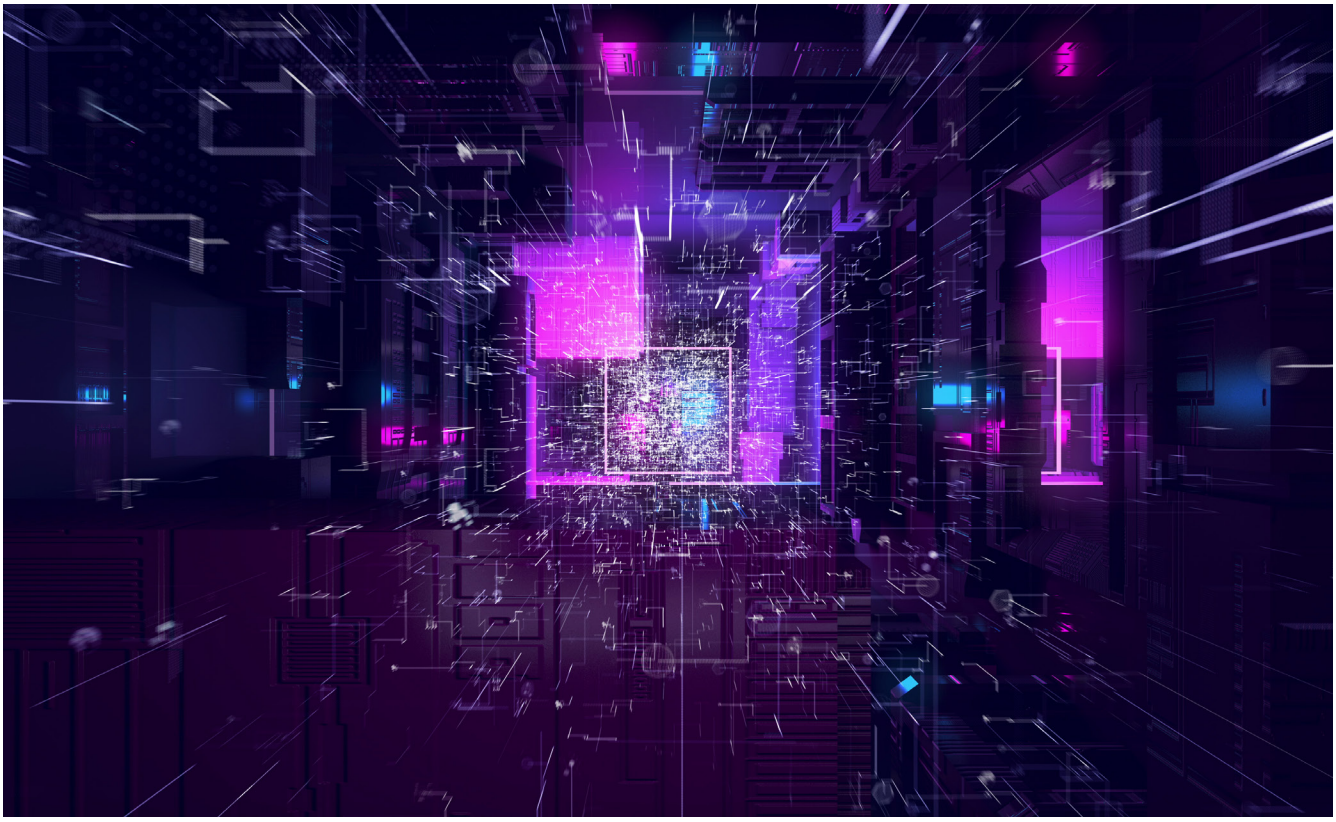
In the article “Securing Cell-Based Architecture in Modern Applications,” Stefania Chaplin discusses how adopting cell-based architecture augments the security landscape of microservice-based systems. The author emphasizes the critical role of the cell router component as the entry point for accessing cells.

In the next article, “Architecting for High Availability in the Cloud with Cellular Architecture,” Chris Price delves into the implementation details of cell-based architecture, covering cell provisioning and management, application deployments, and a DNS-based cell routing mechanism. The author also discusses practical details about security, observability, and cost management aspects relevant to cell-based architectures.

Finally, “Cell-Based Architecture Adoption Guidelines” by Guy Coleman discusses best practices, potential problems, and adoption guidelines for cell-based architectures. The author provides practical advice on how to introduce cell-based architecture into the existing technology platform to avoid common mistakes and allow organizations to fully realize cell-based architecture benefits.

Given the rapid growth of distributed systems and the increasing complexity they bring, there’s no better time than now to consider whether your architecture and organization could benefit from adopting cell-based architecture. Even if you still need more time to fully embrace cell-based architecture, the insights and advice presented in this eMag can serve as a valuable source of thought-provoking information that will inspire you to improve the resiliency of your current and future architectures.

Your feedback is invaluable to us. We eagerly await your thoughts via editors@infoq.com or on X.



How Cell-Based Architecture Enhances Modern Distributed Systems

by **Erica Pisani,** Sr. Software Engineer @Netlify
and **Rafal Gancarz,** Principal Architect & Engineer @Starbucks

The ability to accommodate growth (or scale) is one of the main challenges we face as software developers. Whether you work in a tiny start-up or a large enterprise company, the question of how the system should reliably handle the ever-increasing load inevitably arises when evaluating how to deliver a new product or feature.

The challenges of building and operating modern distributed systems only increase with scale and complexity. Infrastructure resources, in the cloud or on-premises, can experience unexpected and difficult-to-troubleshoot failures that architecture components need to deal with to deliver the required availability.

Monoliths, Microservices, and Resiliency Challenges

Several years ago, microservices and their associated architectures became popular as they helped address some of the scaling challenges that monolithic applications (monorepos) faced.

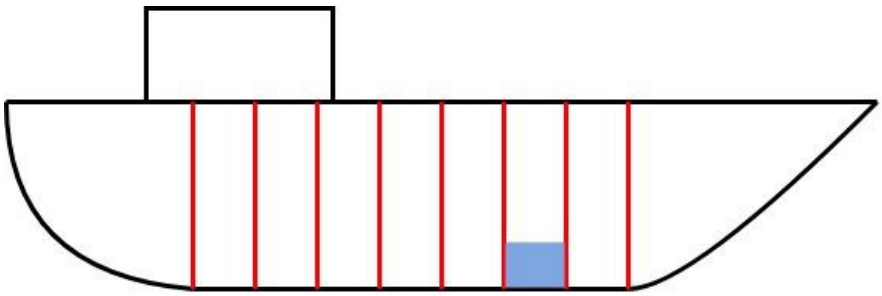
As Susan Fowler mentioned in an interview with InfoQ several years ago, these applications might not support sufficient concurrency or partitioning and, therefore, would reach scalability limitations that lead to performance and stability problems. As these monolithic applications grew,

they became more challenging to work on in local environments. Application deployments became increasingly complex, resulting in teams whose developer velocities would grind to a crawl.

Microservices helped ease those problems by enabling teams to work on, deploy, and scale the services in isolation. However, as with most things, nothing is without flaws, and microservices have their own challenges.

One is that a microservice architecture is very granular down to the level of individual services. As a result, development teams would lack knowledge of where various microservices under their ownership were used in the context of the wider system. It would also be more challenging to know what microservices exist under the ownership of other teams that would be of interest.

These challenges become only more prominent over time as microservices architectures become more complex. Additionally, with the widespread adoption of cloud infrastructure, many companies now manage vast estates of cloud resources, ranging from computing to storage to networking and support services. Any of these resources can experience failures that may result in a minor or significant degradation of service and despite using redundancy and failover mechanisms, some failure modes cannot be fully contained without adopting special measures.



Bulkheads Are Protecting Ships From The Spread of Flooding

The Re-emergence of Cell-Based Architecture

Challenges related to fault isolation are not new and are not specific to microservices or the cloud. As soon as software systems became distributed to accommodate increasing load requirements, many new failure modes had to be considered due to their distributed nature.

Cell-based architectures first emerged during the Service-Oriented Architecture (SOA) era as an attempt to manage failures within large distributed systems and prevent them from affecting the availability of the entire system. These initial implementations by companies like Tumblr, Flickr, Salesforce, or Facebook aimed to limit the blast radius of failures to only a fraction of the customer or user population (a shard) using a self-contained cell as a unit of parallelization for managing infrastructure and application resources and isolating failures.

Cell-based architecture is, firstly, an implementation of the bulkhead pattern, an idea that software engineering adopted from the shipbuilding industry. Bulkheads are watertight vertical partitions in the ship structure that prevent water from flooding the entire ship in case of a hull breach.

For years, the bulkhead pattern has been advertised as one of the key resiliency patterns for modern architectures, particularly microservices. Yet, adoption has been low, mostly due to additional complexity, so most companies choose to prioritize their efforts elsewhere.

Some high-profile companies have recently opted to revisit the cell-based architecture approach to meet the high availability requirements for their microservice-based, cloud-hosted platforms. Slack has migrated most of its critical user-facing services to use a cell-based approach after experiencing partial outages due to AWS availability-zone networking failures. Doordash implemented

the zone-aware routing with its Envoy-based service mesh, moving to an AZ-based cell architecture and reducing cross-AZ data transfer costs. In turn, Roblox is rearranging its infrastructure into cells to improve efficiency and resiliency as it continues to scale.

What these companies have in common is that they run microservice architectures on top of large infrastructure estates in the cloud or private data centers, and they have experienced severe outages due to an unlimited blast radius of infrastructure or application failures. In response, they adopted a cell-based architecture to prevent failures from causing widespread outages.

Amazon Web Services (AWS) has been a long-time adopter and evangelist of cell-based architecture and covered it during its annual re:Invent conference in 2018 and again in 2022. The company also published a whitepaper on cell-based architecture in September 2023.

Building Blocks of Cell-Based Architecture

At a high level, a cell-based architecture consists of the following elements:

- **cells** - self-contained infrastructure/application stacks that provide fault boundaries; responsible for handling application workloads
- **control plane** - responsible for provisioning resources, deploying application services, determining routing mappings, providing platform observability, moving/migrating data, etc.
- **data plane** - responsible for routing the traffic appropriately based on data placement and the cell health (as determined by the control plane)

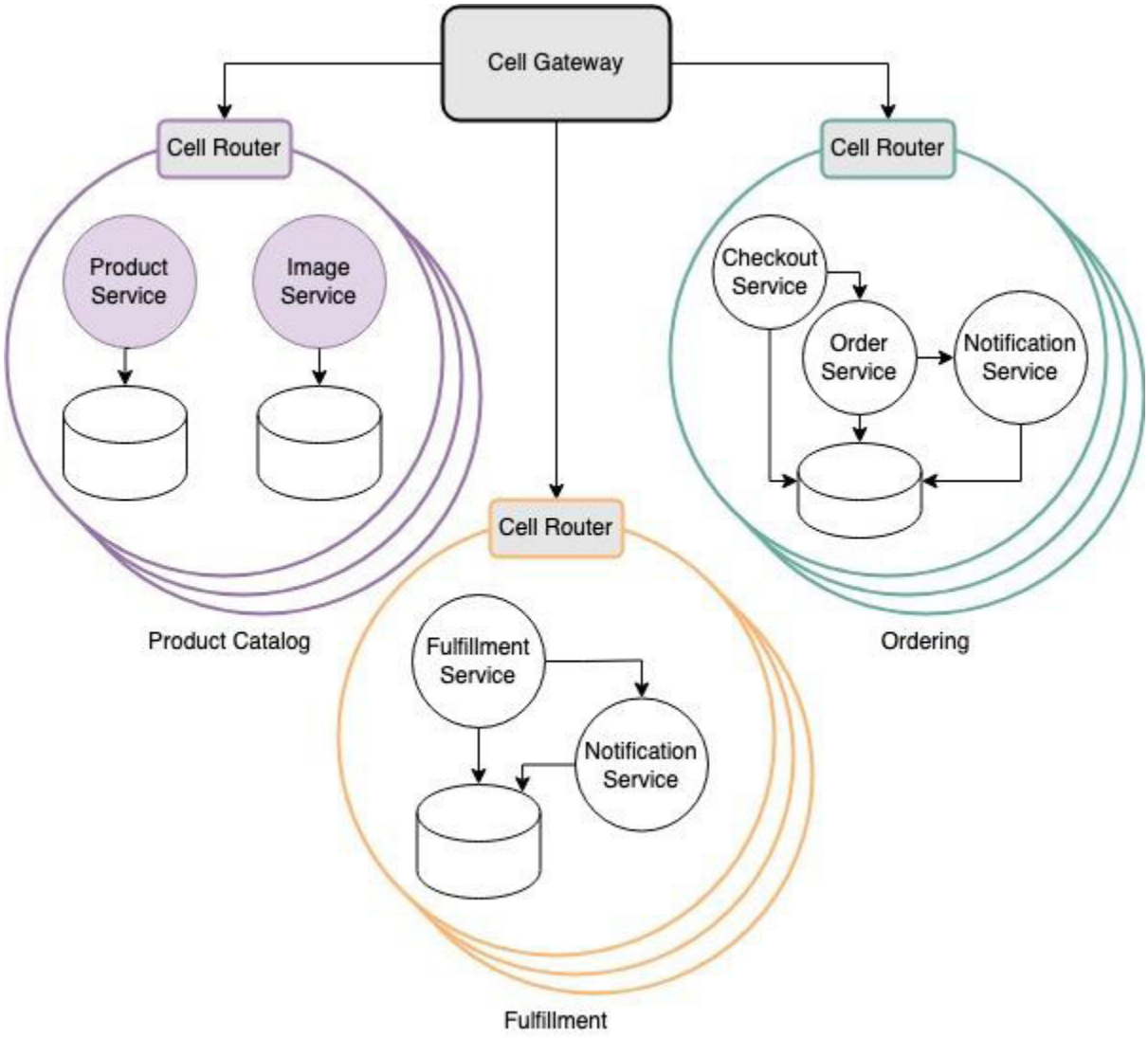
To provide fault-tolerance benefits, cell-based architecture is oriented towards supporting cell-level isolation and low coupling between the control plane and the data plane. It's important to ensure that the data plane can operate without the control plane and should not be directly dependent on the health of the control plane.

Cell as a First-Class Architecture Construct

Adopting cell-based architecture offers an interesting blend of benefits. Cells, first and foremost, provide a fault boundary at the infrastructure level, with cell instances designated to serve a specific segment of the traffic, isolating failures to a subset of a user or customer population. However, they also offer an opportunity to group related application services into domain-specific clusters, aiding with architectural and organizational structures, promoting high cohesion and low coupling, and reducing the cognitive load on the engineering teams.

For small systems or when starting the cell-based architecture adoption effort, it's entirely possible to have a single cell that includes all application services. For larger systems with many application services, multiple cells can be used to organize the architecture along the domain boundaries. Such an approach can help larger organizations adopt the product mindset and align the system architecture with product domains and subdomains. This is particularly important with large microservice systems consisting of hundreds of microservices.

From the fault-tolerance point of view, a cell (or a cell instance) is a complete, independent infrastructure stack that includes all the resources and application service instances required for it to function and serve the workload for the designated segment of traffic (as determined by the cell partitioning strategy). It is crucial to isolate cells as much as possible to keep failures contained. Ideally, cells should be independent of other cells and not share any state or have shared dependencies like databases. Any inter-cell communication should be kept to a minimum; ideally, synchronous API calls should be avoided. Instead, asynchronous, message-



Cell-Based Architecture Combines Domain and Fault-Isolation Boundaries

driven data synchronization should be used. If API interactions can't be avoided, they must go through the cell router so that fault-isolation properties of cell-based architecture are not compromised.

Many considerations regarding cell deployment options include choosing single or multi-DC (data center) deployments and settling on the most optimal cell size. Some organizations adopted cell-based architecture with single DC deployment, where all infrastructure and application resources of a cell instance are co-located in a single data center or availability zone. Such an approach minimizes the impact of gray failures with multi-DC deployments and simplifies health monitoring (cell is either healthy or not). On the other hand, multi-DC deployments, when used appropriately, can provide resiliency in case of DC-level failures, but health monitoring becomes more challenging.

Cell sizing can also play an important role in managing the impact of failures and managing infrastructure costs. Using smaller cells can reduce the

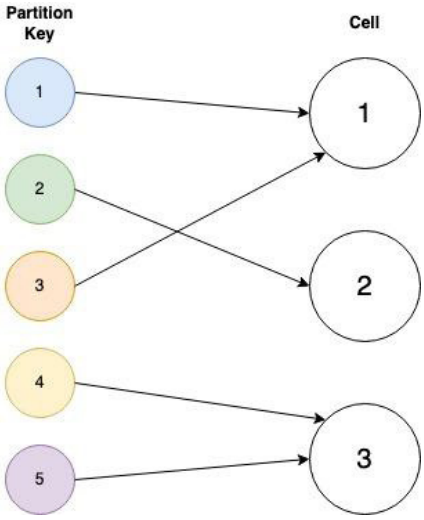
scope of impact (fewer users/customers affected), improve resource utilization (fewer idle resources due to higher cell occupancy levels), and limit the work required to re-route the traffic segment to other cells. However, if the cell size is too small, it may pose challenges in servicing exceptionally large clients/customers, so cells should be big enough to cater to the largest traffic segment based on the partitioning key.

On the other hand, the bigger the cells, the greater the economy of scale in terms of resources, meaning better capacity utilization. Managing fewer cell numbers may be easier on the operational team. Additionally, with larger cell sizes, care needs to be taken to account for infrastructure limits, such as region and account level limits for cloud provider platforms.

Control Plane For Managing Cell-Based Architecture

Adopting the cell-based architecture requires a substantial effort to develop management capabilities that surpass those needed to support regular microservice architecture. Beyond the provisioning and deployment of infrastructure and application services, cell-based architectures require additional functions dedicated to managing and monitoring cells, partitioning and placing the traffic among available cells, and migrating data between cells.

The primary consideration for the cell-based architecture is how to partition traffic between cells, which should be determined separately for each domain using the cell-oriented approach. The first step in working out the optimal partitioning scheme is to choose the partitioning key. In most cases, this may end up being a user or customer identifier, but the choice should be made individually for each case, considering the granularity of traffic segments to avoid segments larger than the chosen cell capacity.



Cell Partitioning Can Use Different Mapping Methods

There are many methods to implement the mapping for cell partitioning, with their respective advantages and disadvantages. These methods range from full mapping, where all mapping records are stored, to using consistent hashing algorithms that offer a fairly stable allocation of items to buckets and minimize churn when adding and removing buckets. Irrespective of the selected mapping approach, it's helpful to provide the override capability to enable special treatment for some partition keys and aid testing activities.

The secondary consideration is the cell placement strategy when new users/customers are onboarded or new cells are provisioned. The strategy should take into account the size and availability capacity of each cell and any cloud provider quotas/limits that can come into play. When the cell-capacity threshold is reached, and a new cell is needed to accommodate the amount of traffic coming to the platform, the control plane is responsible for provisioning the new cell and updating the cell mapping configuration that determines the routing of application traffic by the data plane.

Related to the above is data migration capability, which is important for cell placement (if partition reshuffle is required) or during incidents (if a cell becomes unhealthy and needs to be drained). By their very nature, data migrations are quite challenging from the technical point of view, so this capability is one of the most difficult aspects of delivering cell-based architecture. Conversely, migrating or synching the underlying data between data stores in different cells opens up new possibilities regarding

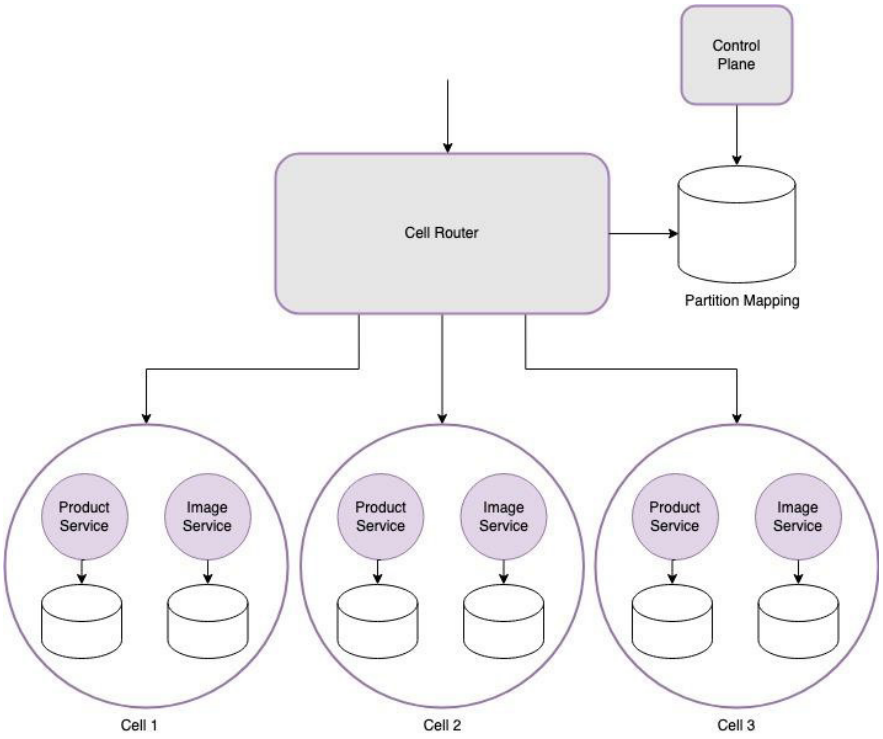
data redundancy and failover, further improving the resiliency offered by adopting cell-based architecture.

Data Plane For Routing Application Traffic

As much as the control plane is responsible for managing the architecture, the data plane reliably moves traffic data around. In the context of cell-based architecture, that translates to routing the traffic to appropriate cells, as determined by the partition mapping records. It's important to emphasize that the routing layer needs to be as simple and horizontally scalable as possible, and complex business logic should be avoided as the data plane is a single point of failure.

The routing layer implementation can employ solutions ranging from DNS and API Gateways to bespoke application services deployed on generic compute or container-based execution platforms. In either case, the partition mapping data has to be available to read from a reliable data store, possibly a highly available distributed database or a blob storage service. The routing layer can support synchronous API calls (HTTP or GRPC) and asynchronous messages, although the latter can be more challenging to implement.

Considering its crucial role in the flow of traffic between cells, the data plane can enforce security policies to ensure that only authorized API requests are served by the services inside cells. As such, a range of security mechanisms can be implemented to protect from unauthorized access, including OAuth or JWT, mutual TLS for authentication, and RBAC or ABAC for authorization.



Cell Router As a Primary Data Plane Component

Benefits of Using Cell-Based Architecture

The primary benefit of adopting cell-based architectures is improved resiliency through fault isolation. Cells provide failure-isolation boundaries and reduce the impact of issues, such as deployment failures, clients abusing the product/platform, operator's mistakes, or data corruption.

Using cells can also help the system's scalability. Ideally, cells should be limited in size to reduce the blast radius of failures, which also makes cells good as a unit for scaling the platform. As the workload increases over time, more cells can be provisioned to cater to the new traffic (new customers/users). Limiting the cell size reduces the risk of surprises from any non-linear scaling factors or unexpected contention points (performance bottlenecks).

Similarly, cells can be used for deployment scoping. Rather than rolling out a new version of service everywhere, organizations

could use canary deployments scoped to a cell (and hence a subset of users/customers) before rolling out the change to a wider user/customer population.

The size-capped cells can be perfect for quantifying the system's performance, as it's easier to test the performance of a single cell and establish the system's scalability characteristics based on scaling out whole cells rather than scaling up components inside cells.

Cells provide the added benefit of grouping services belonging to the same subdomain or bounded context, which can help organizations align team and department boundaries with product domain boundaries. This is particularly relevant for large organizations, where tens or hundreds of teams build and operate large product portfolios.

The last potential benefit could be cost savings from reducing cross-AZ traffic, but this should be weighed against any additional operational costs related to running the routing layer within the data plane.

Considerations for Adopting Cell-Based Architecture

While cell-based architectures offer many advantages in the context of distributed systems, implementing this approach requires additional effort and introduces challenges, so it may not be best suited for every organization like startups still iterating on product-market fit to invest in. Like microservice architectures, cell-based ones require a significant investment in the underlying platform in order to have this architecture speed up your teams' velocity rather than hinder it.

Considering that most companies with non-trivial infrastructure footprint are likely to face challenges that prompted others to adopt cell-based architecture in the past, it may still be worth assessing whether a cell-based approach is worth pursuing.

First and foremost, any company that simply cannot afford widespread outages due to reputational, financial, or contractual requirements should strongly consider adopting cell-based architecture, if not for all, at least for critical user-facing services.

Furthermore, any system where a low Recovery Point Objective (RPO) or Recovery Time Objective (RTO) is required or desired should also consider a cell-based approach. Lastly, multi-tenant products requiring strict infrastructure-level isolation at the tenant level can benefit from cell-based architecture to provide fully dedicated tenancy capabilities.

In any case, the total cost of adopting cell-based architecture should be considered and balanced against expected benefits to determine the anticipated return on investment.

SPONSORED ARTICLE

InfoQ Dev Summit Munich: Learn from German Automotive, Banking, and TelCo Software Practitioners

InfoQ Dev Summit Munich is a two-day in-person software development conference for senior software engineers, architects, and team leaders in the Bavarian capital on September 26th and 27th. The sessions will cover critical topics such as generative AI and platform engineering, with use cases from the German automotive, banking, and telecommunication industries.

For practitioners by practitioners, the conference is designed for senior software engineers, software architects, and team leaders in Europe. What can an attendee expect in Munich? Thanks to 22 technical talks from senior software practitioners, the conference provides actionable insights and practical advice on today's developer priorities with sessions ranging from machine learning to security, from site reliability engineering to scaling Java applications.

Generative AI and bringing machine learning models to production will be major themes of the two days. Olalekan Elesin, engineering director at HRS Group and AWS Machine Learning Hero, will show how to elevate the developer experience with generative AI capabilities on the cloud, while Andrey Cheptsov, founder of dstack, will explain how to leverage open-source LLMs for production. Ines Montani, CEO at Explosion and core developer of spaCy, will demonstrate instead how to take LLMs out of the black box:

I'll show some practical solutions for using the latest state-of-the-art models in real-world applications and distilling their knowledge into smaller and faster components that you can run and maintain in-house.

Performance for system and software architects will be another central topic, with the speed and efficiency of WebAssembly showcased by Danielle Lancashire, principal software engineer at Fermyon. Gunnar Morling, the developer behind the One Billion Row Challenge, will discuss the tricks employed by the fastest solutions:

Parallelization and efficient memory access, optimized parsing routines using SIMD and SWAR, as well as custom map implementations are just some of the topics which we are going to discuss.

The 1BRC went viral within the Java community earlier this year, with results showing that Java can process a file with one billion rows in two seconds. Thanks to sessions by Johannes Bechberger, a JVM developer at SAP, and Markus Kett, CEO at MicroStream, Java developers will also learn how to build a fast firewall with eBPF and create ultra-fast in-memory database applications...

Please read the full-length version of this article here.

Taking Advantage of Cell-Based Architectures to Build Resilient and Fault-Tolerant Systems

by **Yury Niño Roa**, Cloud Infrastructure Engineer @Google

The cell-based architectures have been an emerging paradigm in the last few years, with companies like Slack (which migrated the most critical user-facing services from monolithic to cell-based architectures), Flickr (which employed a federated approach to store the users' data on a shard or cluster of many services), Salesforce (which designed a solution in terms of pods, with functionality self-contained consisting of 50 nodes), and Facebook (which proposed building blocks with services called cells, each cell consisting of a cluster, a metadata store, and controllers in Zookeeper). They have used these architectures to address the challenges of resilience and fault tolerance. The reasons for becoming popular include isolation of failures, improved scalability, simplified maintenance, enhanced fault tolerance, flexibility, and cost-effectiveness.

In the journey to achieve resilience and fault tolerance, the proponents of cell-based architectures have relied on observability, which has played a crucial role in complementing the implementations. This

is the case for Interact, one of the first companies that documented how observability was vital to guaranteeing healthy cell-based architectures. The engineering team of Interact used observability to provide deep insights into system behavior, enabling them to detect issues proactively and facilitating faster recovery from failures. Specifically, they used the maximum number of hosted clients and the maximum number of daily requests per cell to create the new infrastructure alongside the existing architecture.

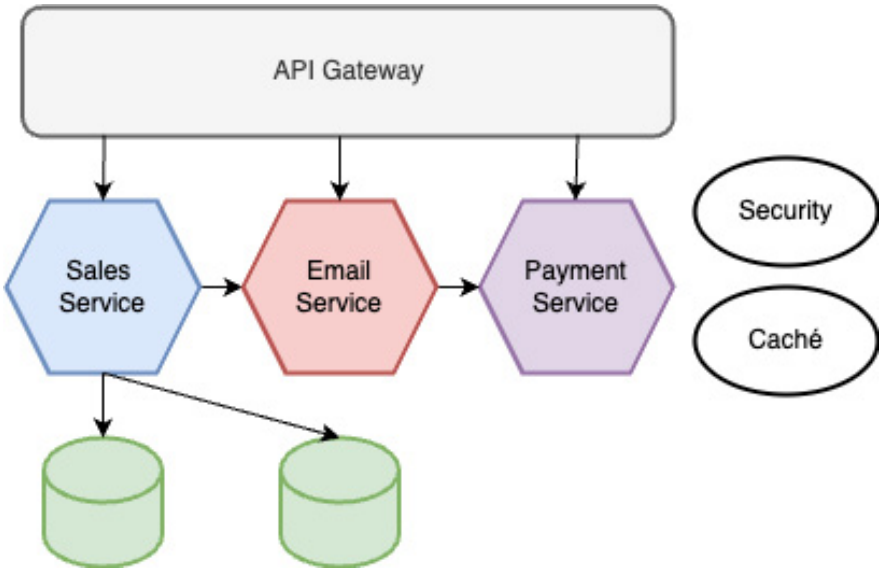


Figure 1. Architectures based on Microservices

This article delves into the resiliency and fault tolerance benefits of adopting cell-based architectures, focusing on the observability aspect. The first part addresses a common question: why use cell-based architectures if microservices are already resilient and fault-tolerant? With that explanation, the second part focuses on observability and the considerations for analyzing inputs and outputs of the cell-based architectures. Finally, it goes over the best practices and takeaways to achieve the visibility needed to detect issues early, diagnose problems quickly, and make informed decisions that enhance resilience and fault tolerance.

Why Use Cell-based Architectures if Microservices are Resilient and Fault Tolerant?

It is a fact that microservices reduce the risk that a single error can bring down the entire system because they use smaller and independently deployable units. This paradigm allows for a failure in a microservice not to affect the whole application. However, it is also a reality that dealing with the complexity due to inter-service communication reduces resilience and fault tolerance levels. While microservices are well-suited to handle large-scale enterprise applications focused on modularity and manageability, cell-based architectures offer advantages in scenarios requiring extreme modularity, scalability, and resource efficiency. It was the reason why Tumblr, which crossed from being a startup to a wildly successful company in a few months, preferred to migrate from a monolith to cell-based architectures and not to microservices. The scalability was a priority for them since they had to evolve their infrastructures while handling huge month-over-month increases in traffic.

Cell-Based Strategy: High Availability for Rapid Growth Requirements

Opting for architectures based on microservices requires carefully analyzing the balance between its advantages and drawbacks. While it offers improved scalability, fault tolerance, and easier operations, it also introduces complexities in implementation and management. However, cell-based architectures are a good fit for systems that prioritize high availability, where experiencing rapid growth is required, or the ability to scale individual components and isolate failures is a priority.

Cell-based architecture is not a universal solution but a strategic choice that aligns with specific business and

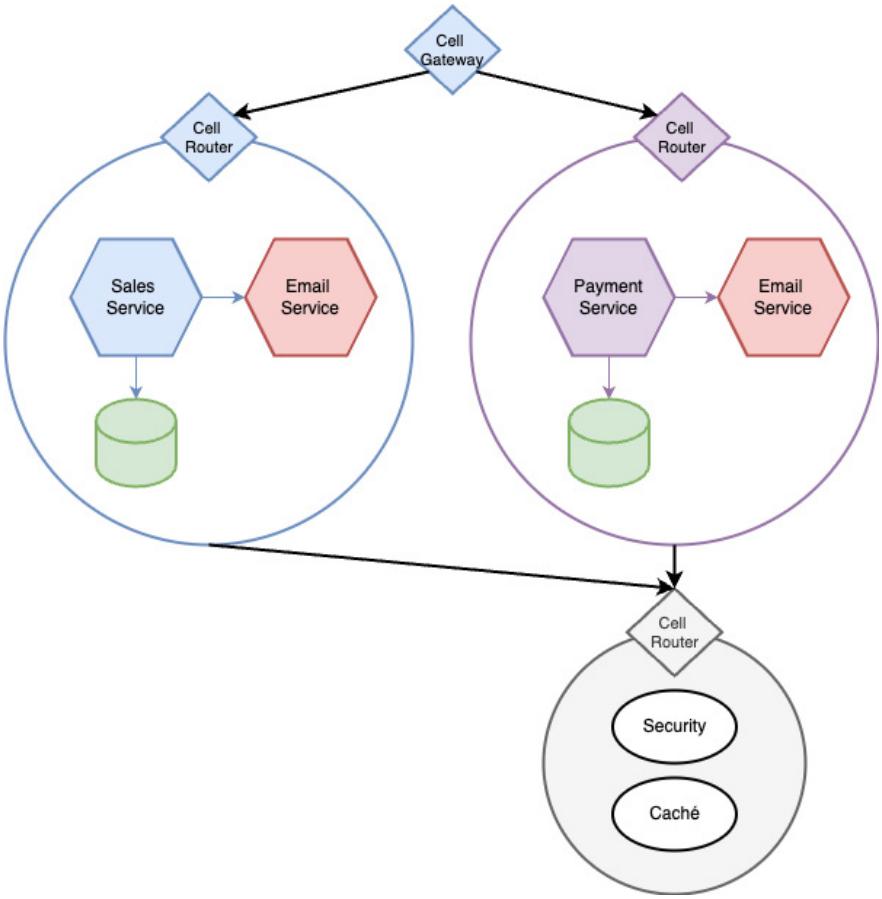


Figure 2. Architectures based on cells

technical needs. Figure 1 illustrates how architectures based on microservices can segregate larger systems into components that wrap up bounded context business domains. Figure 2 shows how cell-based architectures simplify the complexity associated with communication among those services, where each cell is identical and represents an entire stack scaling independently.

Regarding Figure 2, there are two perspectives for implementing cell-based architectures: one in which cells are immutable components that together provide a service, and another in which each cell is identical and represents an entire service. In the first perspective, cells can communicate with each other. In the second, cells are built, deployed, and managed independently as a complete unit since there is no communication among cells.

Cell-based architectures can offer improved resilience and fault tolerance, but how can the operators determine if the system delivers these benefits? The answer is observability.

Considerations to Observe Cell-based Architectures

Cell-based architectures offer a robust approach to building resilient systems. They achieve this through the core principles of isolation, autonomy, and replication. Each cell operates independently, managing its resources and making decisions autonomously. Data and critical services are replicated within the cell for enhanced availability.

These architectures distribute cells across multiple zones or data centers to ensure resilience and fault tolerance, protecting against regional outages. Continuous health checks and monitoring detect failures early, while circuit breakers prevent cascading failures. Load balancing ensures efficient traffic distribution and graceful degradation prioritizes essential functionality during partial failures. Chaos engineering regularly tests resilience by simulating failures.

Observability is the state-of-the-art tool for understanding the state and inner workings of current implementations. Although a system can work without this, collecting, processing, aggregating, and displaying real-time quantitative metrics increases resilience and fault tolerance. That is precisely one of the reasons for including it as a principle in Site Reliability Engineering.

Figure 3. Well-Architected Framework + Observability

Observability is a Pillar of Great Architectures

In addition to being a strategy for understanding the behavior of a system, observability is crucial to achieving the goals of good architecture, particularly in the areas of operational excellence, reliability, and performance efficiency. Figure 3 illustrates the common pillars of well-architected frameworks and gives visibility to their relation to observability. In terms of operational excellence, observability provides the insights needed to understand how systems perform, identify potential problems, and make informed decisions about optimizing them. To achieve performance efficiency, observability enables organizations to identify bottlenecks and inefficiencies in their systems so they can take action to improve performance and reduce costs. Finally, reliability, through monitoring system behavior and detecting anomalies early, observability helps to prevent failures and minimize downtime.

In the path to observe a cell-based architecture, the first step is to define the objectives and identify the metrics such as mean time between failures (MTBF), mean time to repair (MTTR), availability, and recovery time objectives (RTOs), which are well-suited to assess a resilience and fault tolerance level. Once the metrics are clear, the next activity is to provide instrumentation mechanisms incorporating logging, metrics collection, tracing, and event tracking to gather relevant data. A robust infrastructure is then established to collect and aggregate this data efficiently. At this point, observers usually store the collected data in appropriate repositories like time-series databases and process it through filtering, transformation, and enrichment. Analysis tools and visualizations derive insights, identify patterns, and detect anomalies. These insights are integrated into development and operational workflows, establishing feedback loops that drive system design and performance improvements. Finally, the process is iteratively refined, with continuous adjustments to instrumentation, data collection, and analysis based on feedback and evolving requirements. The complete process is illustrated in Figure 4.

Tailoring Observability for Cell-based Architectures

Observability for cell-based architecture requires a tailored approach to address the unique challenges and opportunities presented by this distributed system design. Considering that observability is about monitoring,

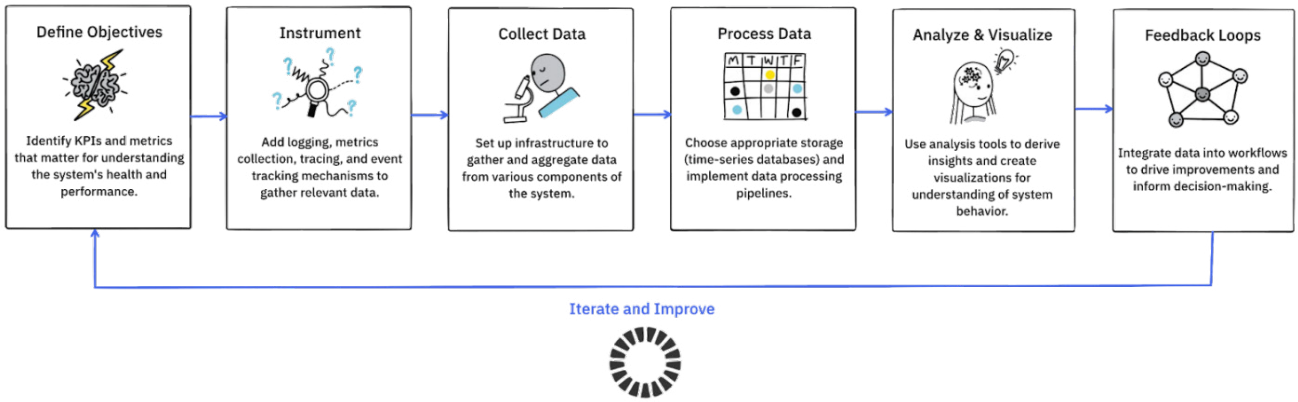


Figure 4. A proposed framework for observability in cell-based architectures.

tracing, and logging, the cell-aware instrumentation includes collecting metrics at the cell level, that is, in general terms, capturing resource utilization (CPU, memory, network), request latency, error rates, and custom business metrics relevant to each cell's function. Distributed tracing is about implementing tracing to track requests across cell boundaries, providing insights into the flow of interactions, and pinpointing bottlenecks. Finally, log aggregation should come from individual cells into a centralized system, allowing for correlation and analysis across the entire architecture.

A second consideration is the creation of cell-level dashboards tailored to each cell's specific functions and KPIs, which enable proper monitoring and troubleshooting. With this configuration, cell-specific alerts based on cell-specific thresholds and anomalies ensure prompt notification of issues affecting individual cells.

A third consideration related to best practices in observability is the need for a unique project that integrates data from various cell-level observability tools into a centralized platform for holistic monitoring and analysis. This makes it easier to leverage the centralized platform to correlate events and metrics across cells, revealing dependencies and potential cascading failures.

A final consideration is cell isolation, which tests individual cells to identify performance bottlenecks and failure modes specific to their functionality. In this consideration, chaos experiments are expected to be designed and developed to allow for controlled disruptions (e.g., network latency, resource constraints) at the cell level to assess resilience and identify weaknesses.

By implementing these practices, organizations can gain deep visibility into the behavior of their cell-based architecture, enabling proactive monitoring, faster troubleshooting, and improved overall system reliability and performance. Always keep in mind that the composition of a cell itself can vary from business to business, which could be an advantage since diversity is precisely one of the benefits of cell-based architectures.

How the Routing Layer Provides Resilience, Fault Tolerance, and Observability

In addition to cells and the control plane, cell routing is crucial in providing resilience and fault tolerance in cell-based architectures. It has the mission to distribute requests to the correct cell based on the partition key, presenting a single endpoint to clients. According to DoorDash, this component offers several benefits, including maintaining

traffic balance, even when services are unevenly distributed across availability zones. This makes it possible to dynamically set traffic weights between pods, eliminating the manual operation and reducing the blast radius of a single or multi-AZ outage, which is critical in fault tolerance and reducing traffic latencies because the caller services connect to more proximal callees.

For reaching fault tolerance in networks, the routing layer uses several mechanisms, which have been documented as innovative solutions for providing resilience. Among them is path redundancy, in which the routing protocols discover and maintain multiple paths to a destination; in that way, if the primary path fails, traffic is automatically rerouted through an alternate path. Another strategy is fast rerouting, designed to detect failures quickly and converge on a new routing solution, minimizing downtime and service disruptions; the classical load balancing that distributes traffic across multiple paths, which prevents congestion and optimizes network resource utilization. And finally, failure detection and recovery where the routing protocol triggers the recovery process to find an alternate path once a failure is detected.

The Role of the Routing Layer in Architecture Observability

The routing layer also significantly impacts observability due to the distributed nature of cell-based systems. As it is a component that centralizes the operation of the cells, it is the best candidate to provide insights into the health and performance of the entire system. Observing the architecture from this component allows traffic patterns, latency, and errors at various points in the network. This allows operators to pinpoint bottlenecks, identify failing components, and optimize routing decisions for better performance.

Furthermore, the routing layer can be instrumented to collect detailed metrics and logs, providing valuable data for troubleshooting and root cause analysis. For instance, tracing the path of a request across multiple cells can reveal where delays occur or errors originate. This granular visibility is essential for maintaining the reliability and efficiency of complex cell-based applications.

In conclusion, the routing layer in a cell-based architecture is not only responsible for directing traffic but also serves as a critical component for observability. Monitoring and analyzing traffic patterns provide valuable insights into the system's behavior, enabling proactive troubleshooting and optimization. This ensures that the cell-based system remains resilient and scalable and performs optimally under varying workloads.

Best Practices to Provide Resilience, Fault Tolerance, and Observability to Cell-based Architectures

Observability in cell-based architectures is crucial for maintaining system health and performance. A fundamental best practice is centralized logging, where logs from all cells are aggregated into a unified repository. This consolidation simplifies troubleshooting and analysis, allowing operators to quickly identify and address issues across the entire system. Structured logging formats further enhance this process by enabling efficient querying and filtering of log data.

Metrics and Monitoring

Metrics and monitoring are equally vital components of observability. Collecting detailed metrics on cell performance, resource utilization, and error rates provides valuable insights into the system's behavior. Setting up dashboards and alerts based on these metrics allows for proactively identifying anomalies and potential bottlenecks. Visualization tools like Grafana can effectively display these metrics, making it easier to spot trends and patterns that may indicate underlying problems.

Distributed Tracing

Distributed tracing is another essential practice for understanding the flow of requests through a cell-based architecture. By tracking requests

as they move across multiple cells, operators can pinpoint performance bottlenecks, latency issues, and failures in microservices interactions. Distributed tracing tools like Jaeger, Zipkin, or AWS X-Ray can help visualize these complex interactions, making diagnosing and resolving problems arising from inter-cell communication more straightforward.

Alerting and Incident Management

Alerting and incident management are integral to a well-rounded observability strategy. Configuring alerts based on predefined thresholds or anomalies in logs and metrics enables timely notifications of potential issues. These alerts can be sent through various channels like email and SMS or integrated into incident management platforms like PagerDuty. Having well-defined incident management processes ensures swift and organized responses to alerts, minimizing downtime and impact on the overall system.

A Holistic Approach to Observability

In addition to these core practices, adopting a holistic approach to observability is beneficial. This includes regularly reviewing and refining logging, monitoring, and tracing configurations to adapt to evolving system requirements. Additionally, incorporating feedback from incident postmortems can help identify areas for improvement in the observability strategy. By continuously enhancing observability, organizations can ensure their cell-based architectures remain resilient, performant, and easy to manage.

Securing Cell-Based Architecture in Modern Applications

by **Stefania Chaplin**, Solutions Architect @GitLab

Cell-based architecture is becoming increasingly popular in the fast-evolving world of software development. The concept is inspired by the design principles of a ship's bulkheads, where separate watertight compartments allow for isolated failures. By applying this concept to software, we create an architecture that divides applications into discrete, manageable components known as cells. Each cell operates independently, communicating with others through well-defined interfaces and protocols.

Cell-based technologies are popular because they provide us with an architecture that is modular, flexible, and scalable. They help engineers rapidly scale while improving development efficiency and enhancing maintainability. However, despite these impressive feats, cell-based technology introduces significant security challenges.

Isolation and Containment

Each cell must operate in a sandboxed environment to prevent unauthorized access to the underlying system or other cells. Containers like Docker or virtual machines (VMs) are often used to enforce isolation. By leveraging sandboxing, even if a cell is compromised, the attacker cannot easily escalate privileges or access other parts of the system.

Permissions and access control mechanisms guarantee that cells can only interact with approved entities. Role-based access control (RBAC) assigns permissions based on roles assigned to users or entities. On the other hand, attribute-based access control (ABAC) considers multiple attributes like user role, location, and time to make access decisions.

Network segmentation is another crucial strategy. Organizations can minimize the attack surface and restrict attackers' lateral movement by creating isolated network zones for different cells. Micro-segmentation takes this a step further by creating fine-grained security zones within the data center, providing even greater control over network traffic. Enforcing strict access controls and monitoring traffic within each segment enhances security at the cell level, helping meet compliance and regulatory requirements by guaranteeing a robust and secure architecture.

Zero-Trust Security

In a cell-based architecture, adopting a zero-trust approach means treating every interaction between cells as potentially risky, regardless of origin. This approach requires constantly verifying each cell's identity and applying strict access controls. Trust is always earned, never assumed.

Zero trust involves explicitly checking each cell's identity and actions using detailed data points. It means limiting access so cells only have the permissions they need (least privilege access) and creating security measures that assume a breach could already be happening.

To implement zero trust, enforce strong authentication and authorization for all cells and devices. Use network segmentation and micro-segmentation to isolate and contain any potential breaches. Employ advanced threat detection and response tools to quickly spot and address threats within the architecture. This comprehensive strategy ensures robust security in a dynamic environment.

Authentication and Authorization

Strong authentication mechanisms like OAuth and JWT (JSON Web Tokens) verify cells' identities and enforce strict access controls. OAuth is a widely used framework for token-based authorization. It allows secure resource access without sharing credentials, which is particularly useful in distributed systems. This framework lets cells grant limited access to their resources to other cells or services, reducing the risk of credential exposure.

JWTs, on the other hand, are self-contained tokens that carry claims about the user or system, such as identity and permissions. They provide a compact and secure way to transmit information between cells. Signed with a cryptographic algorithm, JWTs ensure data authenticity and integrity. When a cell receives a JWT, it can verify the token's signature and decode its payload to authenticate the sender and authorize access based on the claims in the token.

Using OAuth for authorization and JWTs for secure information transmission achieves precise access control in cell-based architectures. As a result, cells only access the resources they are permitted to use, minimizing the risk of unauthorized access. Furthermore, these mechanisms support scalability and flexibility. Cells can dynamically issue and validate tokens without needing a centralized authentication system, enhancing the overall security and efficiency of the architecture and making it robust and adaptable.

Encryption

Encryption ensures that only the intended recipient can read the data. Hashing algorithms verify that the data hasn't been altered during transmission, and certificates with public-key cryptography confirm the identities of the entities involved. All data exchanged between cells should be encrypted using strong protocols like TLS. Using encryption prevents eavesdropping and tampering and keeps the data confidential, intact, and authenticated. It also protects sensitive information

from unauthorized access, ensures data integrity, and verifies the identities of the communicating parties.

Organizations should follow best practices to implement TLS effectively. It's crucial to ensure the TLS implementation is always up to date and robust by managing certificates properly, renewing them before they expire, and revoking them if compromised. Additional security measures include enabling Perfect Forward Secrecy (PFS) to keep session keys secure, even if the server's private key is compromised. In order to avoid using deprecated protocols, it's essential to check and update configurations regularly.

Mutual TLS (mTLS)

mTLS (mutual TLS) boosts security by ensuring both the client and server authenticate each other. Unlike standard TLS, which only authenticates the server, mTLS requires both sides to present and verify certificates, confirming their identities. Each cell must present a valid certificate from a trusted Certificate Authority (CA), and the receiving cell verifies this before establishing a connection. This two-way authentication process ensures that only trusted and verified cells can communicate, significantly reducing the risk of unauthorized access.

In addition to verifying identities, mTLS also protects data integrity and confidentiality. The encrypted communication channel created by mTLS prevents eavesdropping and tampering, which is crucial in cell-based architectures where sensitive data flows between many components. Implementing mTLS involves managing certificates for all cells. Certificates must be securely stored, regularly updated, and properly revoked if compromised. Organizations can leverage automated tools and systems to assist in managing and renewing these certificates.

Overall, mTLS ensures robust security by establishing mutual authentication, data integrity, and confidentiality. It provides an additional layer of security to help maintain the trustworthiness and reliability of your system and prevent unauthorized access in cell-based architectures.

API Gateway

An API gateway is a vital intermediary, providing centralized control over API interactions while simplifying the system and boosting reliability. By centralizing API management, organizations can achieve better control, more robust security, efficient resource usage, and improved visibility across their architecture.

An API Gateway can be one of the best options for cell-based architecture for implementing the cell router. A single entry point for all API interactions with a given cell reduces the complexity of direct communication between numerous microservices and reduces the surface area exposed to external agents. Centralized routing makes updating, scaling, and ensuring consistent and reliable API access easier. The API gateway handles token validation, such as OAuth and JWT, verifying the identities of communicating cells. It can also implement mutual TLS (mTLS) to authenticate the client and server. Only authenticated and authorized requests can access the system, maintaining data integrity and confidentiality.

The API gateway can enforce rate limiting to control the number of requests a client can make within a specific time frame, preventing abuse and ensuring fair resource usage. It is critical for protecting against denial-of-service (DoS) attacks and managing system load. The gateway also provides comprehensive logging and monitoring capabilities. These capabilities offer valuable insights into traffic patterns, performance metrics, and potential security threats, which allows for proactive identification and resolution of issues while maintaining system robustness and efficiency. Effective logging and monitoring, facilitated by the API gateway, are crucial for incident response and overall system health.

Service Mesh

A service mesh helps manage the communication between services. It handles the complexity of how cells communicate, enforcing robust security policies like mutual TLS (mTLS). Data is encrypted in transit, and the client and server are verified during every transaction. Only authorized services can interact, significantly reducing the risk of unauthorized access and data breaches.

They also allow for detailed access control policies, precisely regulating which services can communicate with each other, further strengthening the security of the architecture. Beyond security, a service mesh enhances the visibility and resilience of cell-based architectures by providing consistent logging, tracing, and monitoring of all service interactions. This centralized view enables the real-time detection of anomalies, potential threats, and performance issues, facilitating quick and effective incident responses.

The mesh automatically takes care of retries, load balancing, and circuit breaking. Communication remains reliable even under challenging conditions, maintaining the availability and integrity of services. A service mesh simplifies security management by applying security and operational policies consistently across all services without requiring changes to the application code, making it easier to enforce compliance and protect against evolving threats in a dynamic and scalable environment. Service meshes secure communication and enhance the overall robustness of cell-based architectures.

Doordash recently shared that it implemented zone-aware routing using the Envoy-based service mesh. The solution allowed the company to efficiently direct traffic within the same availability zone (AZ), minimizing more expensive cross-AZ data transfers.

Centralized Registry

A centralized registry is the backbone for managing service discovery, configurations, and health status in cell-based architectures. By keeping an up-to-date repository of all cell and service instances and their metadata, we can ensure that only registered and authenticated services can interact. This centralization strengthens security by preventing unauthorized access and minimizing the risk of rogue services infiltrating the system. Moreover, it enforces uniform security policies and configurations across all services. Consistency allows best practices to be applied and reduces the likelihood of configuration errors that could lead to vulnerabilities.

In addition to enhancing access control and configuration consistency, a centralized registry significantly improves monitoring and incident response capabilities. It provides real-time visibility into the operational status and health of services. Allowing the rapid identification and isolation of compromised or malfunctioning cells. Such a proactive approach is crucial for containing potential security breaches and mitigating their impact on the overall system.

The ability to audit changes within a centralized registry supports compliance with regulatory requirements and aids forensic investigations. Maintaining detailed logs of service registrations, updates, and health checks strengthens the security posture of cell-based architectures. Through such oversight, cell-based architectures remain resilient and reliable against evolving threats.

Cell Health

Keeping cells healthy allows each cell to run smoothly and reliably, which, in turn, maintains the system's overall integrity and security. Continuous health monitoring provides real-time insights into how each cell performs, tracking important metrics like response times, error rates, and resource use. With automated health checks, the system can quickly detect any anomalies, failures, or deviations from expected performance. Early detection allows for proactive measures, such as isolating or shutting down compromised cells

before they can affect the broader system, thus preventing potential security breaches and ensuring the stability and reliability of services.

Maintaining cell health also directly supports dynamic scaling and resilience, essential for strong security. Healthy cells allow the architecture to scale efficiently to meet demand while keeping security controls consistent. When a cell fails health checks, automated systems can quickly replace or scale up new cells with the proper configurations and security policies, minimizing downtime and ensuring continuous protection. This responsive approach to cell health management reduces the risk of cascading failures. It improves the system's ability to recover quickly from incidents, thereby minimizing the impact of security threats and maintaining the overall security posture of the architecture.

Infrastructure as Code

Infrastructure as Code (IaC) enables consistent, repeatable, and automated infrastructure management. By defining infrastructure through code, teams can enforce standardized security policies and configurations across all cells from the start, enforcing best practices and compliance requirements.

Tools like Terraform or AWS CloudFormation automate provisioning and configuration processes, significantly reducing the risk of human error, a common source of security vulnerabilities. A consistent setup helps maintain a uniform security posture, making it easier to systematically identify and address potential weaknesses.

IaC also enhances security through version control and auditability. All infrastructure configurations are stored in version-controlled repositories. Teams can track changes, review configurations, and revert to previous states if needed. This transparency and traceability are critical for compliance audits and incident response, providing a clear history of infrastructure changes and deployments.

IaC facilitates rapid scaling and recovery by enabling quick and secure provisioning of new cells or environments. Even as the architecture grows and evolves, security controls are consistently applied. In short, IaC streamlines infrastructure management and embeds security into the core of cell-based architectures, boosting their resilience and robustness against threats.

Conclusion

Securing cell-based architecture is essential to fully capitalize on its benefits while minimizing risks. To achieve this, comprehensive security measures must be put in place. Organizations can start by isolating and containing cells using sandbox environments and strict access control mechanisms like role-based and attribute-based access control. Network segmentation and micro-segmentation are crucial — they minimize attack surfaces and restrict the lateral movement of threats.

Adopting a zero-trust approach is vital. It ensures that each cell's identity is continuously verified. Robust authentication mechanisms such as OAuth and JWT and encrypted communication through TLS and mTLS protect data integrity and confidentiality. A service mesh handles secure, reliable interactions between services. Meanwhile, a centralized registry ensures that only authenticated services can communicate, boosting monitoring and incident response capabilities.

API gateways offer centralized control over API interactions, ensuring consistent security between cells. Continuous health monitoring and Infrastructure as Code (IaC) further enhance security by automating and standardizing infrastructure management, allowing rapid scaling and recovery.

By integrating these strategies, organizations can create a robust security framework that allows cell-based architectures to operate securely and efficiently in today's dynamic technology landscape.

Architecting for High Availability in the Cloud with Cellular Architecture

by **Chris Price**, Software Engineer @Momento

What is Cellular Architecture?

Cellular architecture is a design pattern that helps achieve high availability in multi-tenant applications. The goal is to design your application so that you can deploy all of its components into an isolated "cell" that is fully self-sufficient. Then, you create many discrete deployments of these "cells" with no dependencies between them.

Each cell is a fully operational, autonomous instance of your application ready to serve traffic with no dependencies on or interactions with any other cells.

Traffic from your users can be distributed across these cells, and if an outage occurs in one cell, it will only impact the users in that cell while the other cells remain fully operational. This minimizes the "blast radius" of any outages your service may experience and helps ensure that the outage doesn't impact your SLA for most of your users.

There are many different strategies for organizing your cells and deciding which traffic should be routed to which cell. For more information on the benefits of cellular architecture

and some examples of these different strategies, we highly recommend Peter Voshall's talk from re:Invent 2018: "How AWS Minimizes the Blast Radius of Failures".

Managing an application with many independent cells can be daunting. For this reason, it's extremely valuable to build as much automation as possible for the common infrastructure tasks necessary for creating and maintaining cells. For the rest of this article, we will focus less on the "why" of cellular architecture and more on the "how" of creating this automation. For more info on the "why," check out Peter's talk and the links in the Additional Resources section at the end of the article!

Automating Your Cellular Architecture

In the quest to automate cellular infrastructure, there are five key problems to solve:

- Isolation: how do we ensure distinct boundaries between our cells?
- New cells: how do we bring them online consistently and efficiently?
- Deployment: how do we get the latest code changes into each cell?
- Permissions: how do we ensure the cell is secure and manage both its inbound and outbound permissions effectively?
- Monitoring: how does an operator determine the health of all cells at a glance and easily identify which cells are impacted by an outage?

There are many tools and strategies that can be used to address these problems. This article will discuss the tools and solutions that have worked well for us at Momento.

Before solving these specific problems, though, we will talk a bit about standardization.

Standardization

Standardizing certain parts of the build/test/deploy life cycle for all of the components in your application makes it much easier to build general-purpose automation around them. And generalizing your automation will make it much easier to re-use your infrastructure code across all the components you need to deploy to each cell.

It's important to note that we are discussing **standardization**, not **homogenization**. Most modern cloud applications are not homogenous. Your application may comprise five different microservices running on some combination of platforms such as Kubernetes, AWS Lambda, and EC2. To build general-purpose automation for all these different types of components, we just need to standardize a few specific parts of their life cycle.

Standardization - Deployment Templates

So, what do we need to standardize? Let's think about the steps typically involved in rolling out a code change to the production environment. They might be something like this:

- A developer **commits** code changes to the version control repository.
- We **build** a binary artifact with the latest changes; this might be a docker image, a JAR file, a ZIP file, or some other artifact.
- The artifact is **published** or **released**: the docker image is pushed to a docker repository, the JAR file is pushed to a maven repository, the ZIP file is pushed to some location in cloud storage, etc.
- The artifact is **deployed** to your production environment(s). This typically involves serially deploying to each of your cells in a cellular environment.

So, for any given component of our application, this is a rough template of what we want the deployment process to look like:

Now, one of the goals of cellular architecture is to minimize the blast radius of outages, and one of the most likely times that an outage may occur is immediately after a deployment. So, in practice, we'll want to add a few

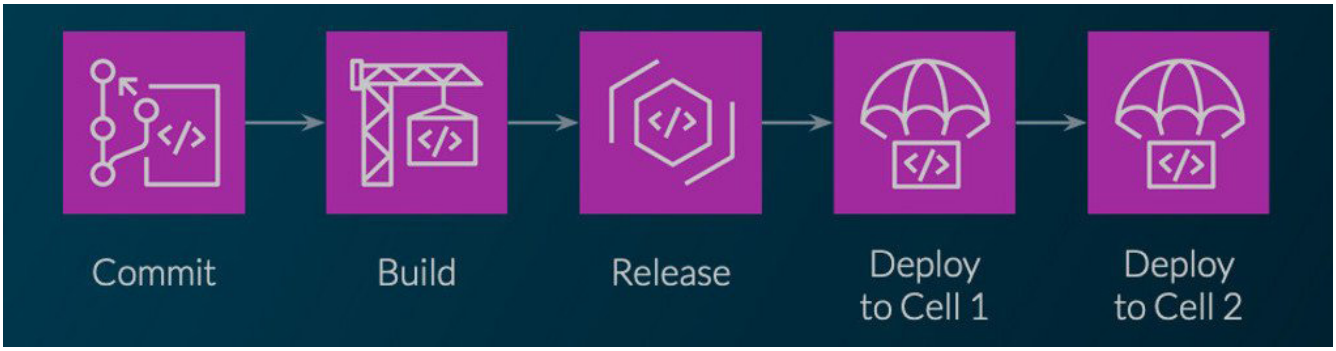


Figure 1: Minimal deployment template

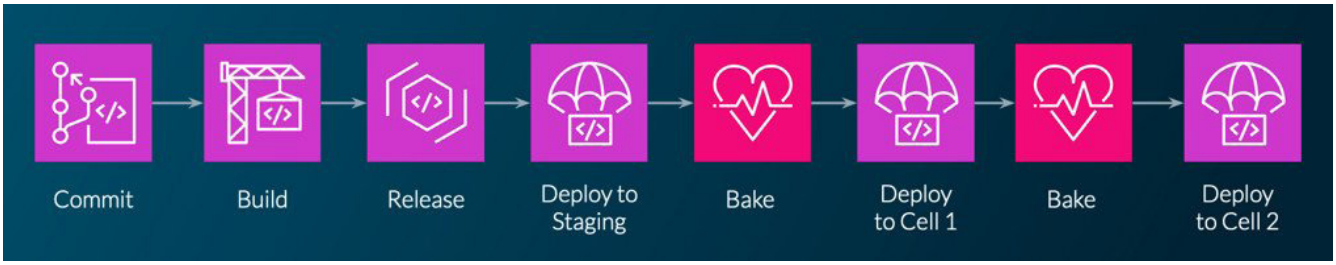


Figure 2: Deployment template with "bake" stages

protections to our deployment process so that if we detect an issue, we can stop deploying the changes until we've resolved it. To that end, adding a "staging" cell that we can deploy to first and a "bake" period between deployments to subsequent cells is a good idea. During the bake period, we can monitor metrics and alarms and stop the deployment if anything is awry. So now our deployment template for any given component might look more like this:

Now, our goal is to generalize our automation so that it's easy to achieve this set of deployment steps **for any application component, regardless of the technology that the component is built on**.

Many tools can automate the steps above. For the rest of the article, we will use some examples based on the tools we've chosen at Momento, but you can achieve these steps using whatever tools are best suited to your particular environment.

At Momento, most of our infrastructure is deployed in AWS, so we have leaned into the AWS tools. So, for a given component of our application that runs on EC2 and is deployed via CloudFormation, we use:

- AWS CodePipeline for defining and executing the stages
- AWS CodeBuild for executing individual build steps
- AWS Elastic Container Registry for releasing the new docker image for the component
- AWS CloudFormation for deploying the new versions to each cell
- AWS Step Functions to monitor alarms during the "bake" step and decide whether it's safe to deploy to the next cell

For a Kubernetes-based component, we can achieve the same steps with only a minor variation: instead of CloudFormation, we use AWS Lambda to make API calls to k8s to deploy the new image to the cells.

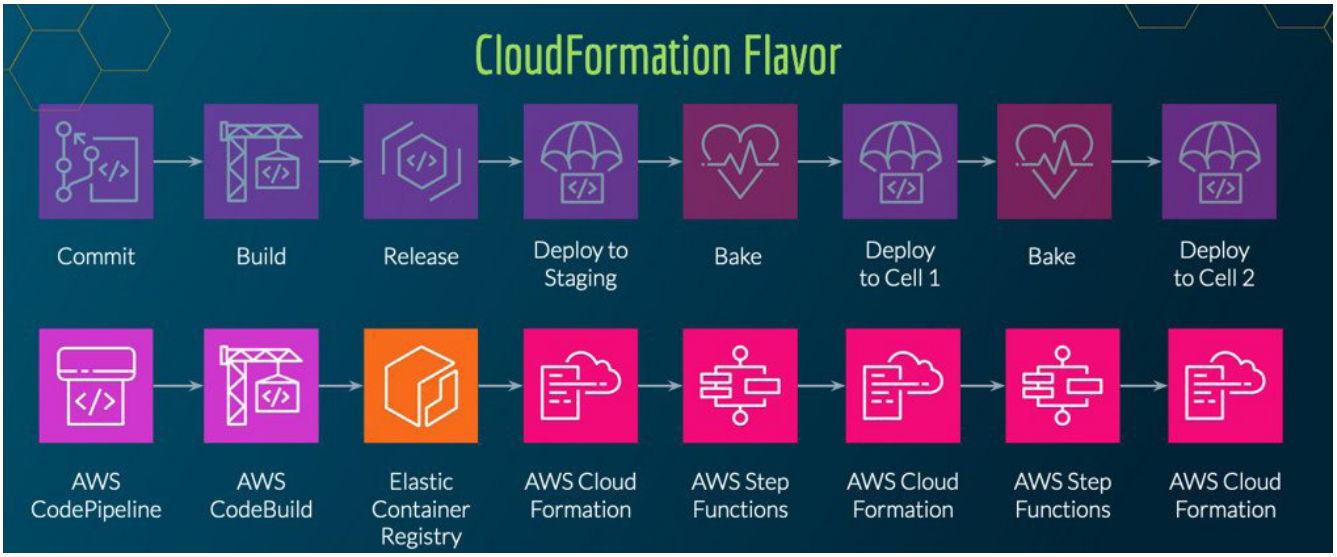


Figure 3: Deployment Stages Implementation - CloudFormation Flavor

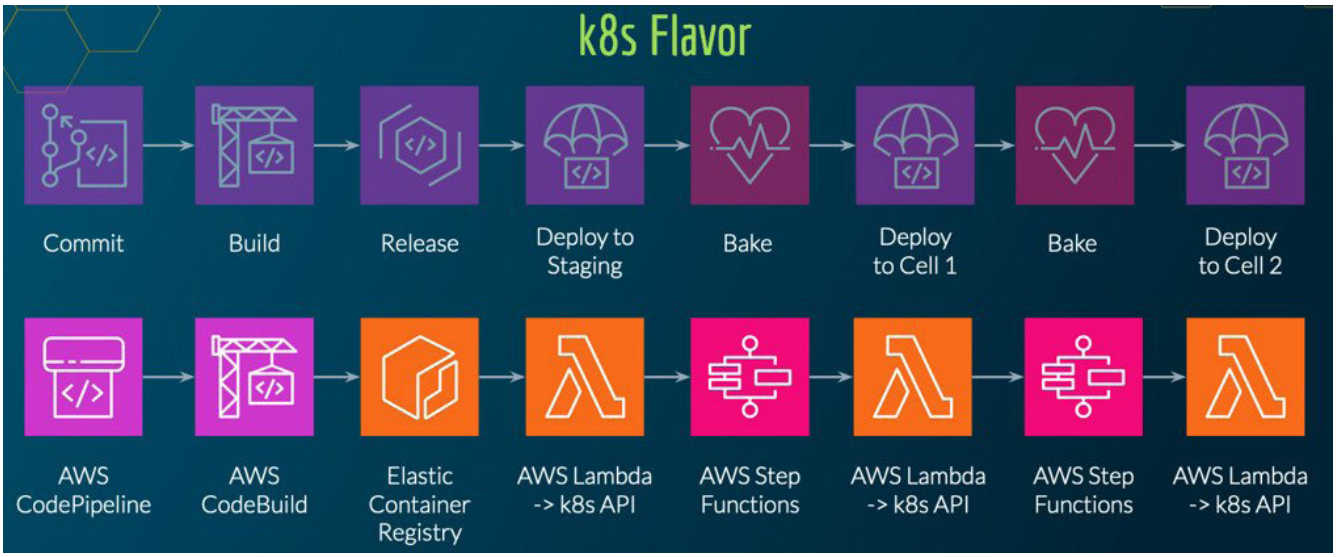


Figure 4: Deployment Stages Implementation - Kubernetes Flavor

Hopefully, you see what we're working toward here; despite differences in technology stacks for the components that make up our application, we can define a general-purpose template for what steps are involved in rolling out a new change. Then, we can implement those steps mainly using the same toolchain, with minor modifications for specific steps. Standardizing a few things about the build life cycle across all of our components will allow us to build automation for these steps in a general way, meaning that we can reuse a lot of infrastructure code and that our deployments will be consistent and recognizable across all of our components.

Standardization - Build Targets

So, how do we standardize the necessary steps across our various components? One valuable strategy is to define some standardized build targets and reuse them across all components. At Momento, we use a tried-and-true technology for this: Makefiles.

Service 1

```
pull-request-ci: gradle-build build-common-ts
  cd infrastructure/aws \
    && ./ci-synth.sh \
    && ./ci-gcp-control-account-synth.sh \
    && cd ../gcp \
    && ./ci-synth.sh

pipeline-build: gradle-build build-common-ts
  cd infrastructure/aws \
    && npm ci --verbose \
    && npm run build

cell-bootstrap: build-common-ts
  cd infrastructure/aws \
    && ./deploy-stacks.sh
```

Service 2

```
pull-request-ci:
  cargo fmt -- --check \
    && cargo clippy --all-targets --all-features \
    && cargo build --frozen \
    && cargo test -- --nocapture

pipeline-build:
  cargo build --locked --release \
    && ./package.sh arm64v8/router.tar.gz \
    && ./package.sh arm64v8/bulkingest.tar.gz \

cell-bootstrap:
  cd infrastructure/aws \
    && ./release-docker-image.sh

gcp-cell-bootstrap:
  cd infrastructure/gcp \
    && ./release-docker-image.sh
```

Figure 5: Standardized Build Targets using Makefiles

Makefiles are pretty simple, and they've been around forever. They work for this purpose just fine.

On the left, you can see a snippet from a Makefile for one of our Kotlin microservices. On the right is a snippet from a Makefile for a Rust service. The build commands are very different, but the important point is that we have the same exact list of targets on both of these.

For example, we have a **pipeline-build** target, which controls what happens in the **build** step of the deployment process for that particular service. Then, we have some targets for a «cell bootstrap» and a «GCP cell bootstrap» because we can deploy to either AWS cells or GCP cells for Momento. The Makefile target names are the same; what this means is that other pieces of our infrastructure that are operating outside of these individual services now have this common lifecycle that they know they can rely on the existence of inside of each of the components that they need to interact with when we're doing things like deploying.

Standardization - Cell Registry

Another building block that helps us standardize our automation is a "cell registry." This is just a mechanism for giving us a list of all the cells we've created and the essential bits of metadata about them.

At Momento, we built our cell registry in TypeScript. We have about 100 lines of TypeScript that define a few simple interfaces that we can use to represent all of the data about our cells. We have a CellConfiguration interface, which is probably the most important one; it captures all the vital information about a given cell. Is this a prod cell or a developer cell? What region is it in? What are the DNS names of the endpoints in this cell? Is it an AWS cell or a GCP cell?

We also have a MomentoOrg interface, which contains an array of CellConfigurations. So, the org is just a way to keep track of all the existing cells.

Using the model provided by these interfaces, we can write just a bit more typescript code to instantiate them and create all

```
5+ usages  new *
export interface DnsConfig {
  domainName: string;
  nameServers: string[];
}

1 usage  new *
export interface MomentoCellConfiguration {
  momentoOrgName: MomentoSsoOrgName;
  momentoCellScale: MomentoCellScale;
  region: string;
  dnsConfig?: DnsConfig;
  cloudProviderConfig: MomentoCloudProviderConfig;
}
```

```
2 usages  new *
export interface MomentoSsoOrg {
  name: string;
  id: string;
  rootAccountOrgArn: string;
  developers: Array<MomentoSsoDeveloper>;
  cicdSharedAccount: MomentoSsoAccount;
  cellAccounts: Array<MomentoSsoCellAccount>;
  networkDnsAccount: MomentoSsoNetworkDnsAccount;
  sharedMonitoringAccount: MomentoSharedMonitoringAccount;
}
```

Figure 6: TypeScript model for cell registry

```
cellAccounts: [
  {
    name: 'cell-alpha',
    id: '902102490210',
    cells: [
      {
        momentoOrgName: PREPROD,
        momentoCellScale: PROD,
        region: 'us-west-2',
        dnsConfig: {
          domainName: 'alpha.staging.momento.com',
          nameServers: [
            'ns-90210.awsdns-42.co.uk',
            'ns-90210.awsdns-42.com',
            'ns-90210.awsdns-42.org',
            'ns-90210.awsdns-42.net',
          ],
        },
        cloudProviderConfig: {
          cloudProvider: AWS,
        },
```

Figure 7: Cell registry data for our ‘alpha’ cell

the actual data about our cells. Here’s a snippet from that code:

This is what the data might look like for our “alpha” cell. We’ve got a cell name, an account ID, the region, the DNS config, etc. Now, whenever we want to add a new cell, we can just enter this cell registry code and add a new entry to this array.

Now that we have all of this data about our cells, we need to publish it somewhere that we can make it accessible from the rest of our infrastructure. Depending on your needs, you might do something sophisticated, like putting the data into a database you can query. We don’t need anything sophisticated, so we just store the data as JSON on S3.

The final component of the cell registry is a small TypeScript library that knows how to retrieve this data from S3 and convert it back into a TypeScript object. We

```
const cellInfo = await getCellForAccount(accountInfo);
const gitRepos : GitRepoInfo[]  = await prepareGitClones(allRequiredRepos);
for (const repo : GitRepoInfo  of gitRepos) {
  runMakeTargetForRepo(cellInfo, repo, makeTarget: 'cell-bootstrap');
}
```

Figure 8: Cell bootstrap script

publish that library to a private npm repository, and we can consume it from anywhere else in our infrastructure code. This allows us to start building some generalization patterns across all of our infrastructure automation; we can loop over all the cells and configure the same automation for each one.

Standardization - Cell Bootstrap Script

The last piece of standardization we use to generalize our automation is a "cell bootstrap script." Deploying all of the components of an application to a new cell can be very challenging, time-consuming, and error-prone. However, a cell bootstrap script can simplify the process and ensure consistency from one cell to the next.

Assuming that the source code for each of your application components lives in a git repo, then, given the building blocks above, the logic of bootstrapping a new cell is as simple as this:

- Look up all the cell metadata that we need for this cell (e.g., AWS account ID, DNS configuration, etc.) using the cell registry
- For each application component:
  - Clone the git repo for that component
  - Run the standardized cell-bootstrap target from the Makefile

With just five lines of code, this script offers a generic and extensible solution for deploying new cells of the application. If you introduce new components to your application, the script remains adaptable and ensures a straightforward and consistent deployment process.

Putting It All Together

Now that we've defined some standardized building blocks to help us organize the information about our cells and generalize various life cycle tasks for our application components, it is time to revisit our list of the five problems we need to solve to automate our infrastructure across all our cells.

Isolation

The easiest way to ensure isolation between your cells in an AWS environment is to create a separate AWS account for each cell. At first, this may seem daunting if you are not used to managing multiple accounts. However, AWS's tooling today is very mature, making it easier than you might imagine.

Deploying a cell within a dedicated AWS account ensures isolation from your other cells as the default posture. You would have to set up complex cross-account IAM policies for one cell to interact with another. Conversely, if you deploy multiple cells into a single AWS account, you will need to set up complex IAM policies to *prevent* the cells from being able to interact with one another. IAM policy management is one of the most challenging parts of working with AWS, so any time you can avoid the need to do so, that will save you time and headaches.

Another benefit of using multiple accounts is that you can link the accounts together using AWS

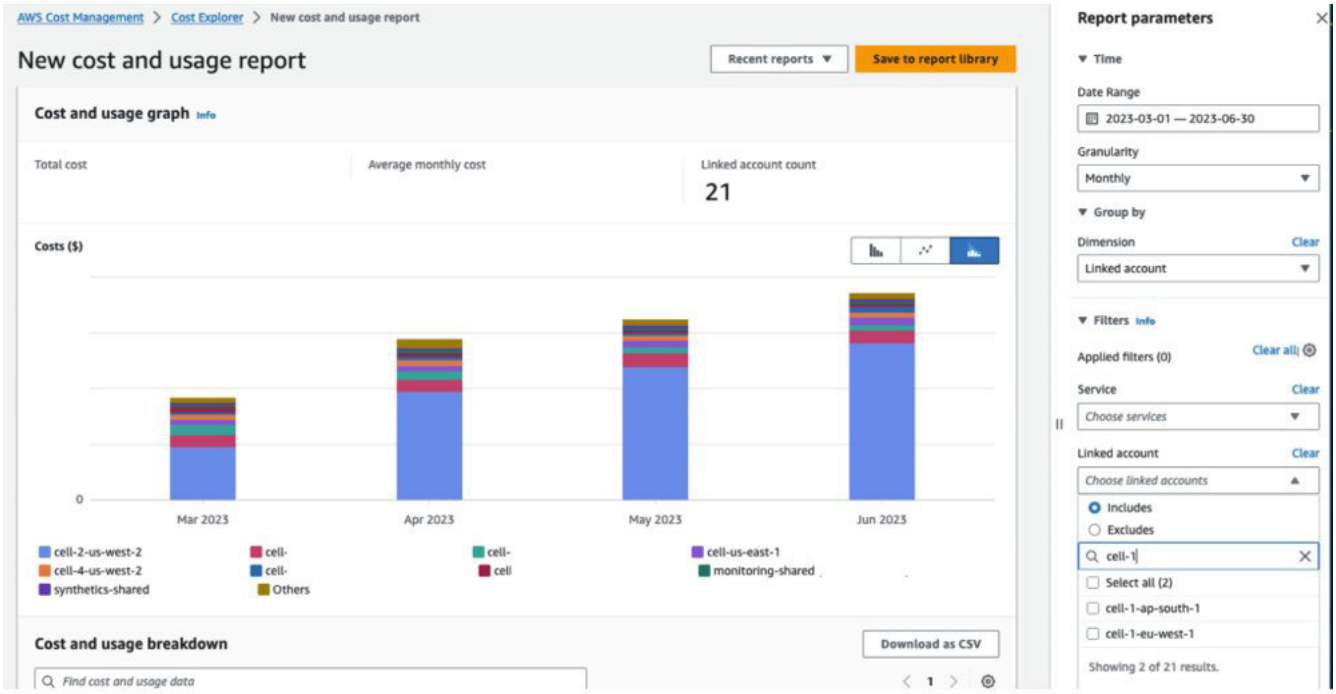


Figure 9: AWS Cost Explorer view of costs for each cell account

Organizations and then visualize and analyze your costs by cell using AWS Cost Explorer. If you instead choose to deploy multiple cells into a single AWS account, you must carefully tag the resources associated with each cell to view per-cell costs. Using multiple accounts allows you to get this for free.

One challenge that goes hand-in-hand with cellular architecture is routing. When you have multiple isolated cells, each running a copy of your application, you must choose a strategy for routing traffic from your users to the desired cells.

If your users interact with your application through an SDK or other client software that you provide, then one easy way to route traffic to individual cells is to use unique DNS names for each cell. This is the approach we use at Momento; when our users create authentication tokens, we include the DNS name for the appropriate cell for that user as a claim inside of the token, and then our client libraries can route the traffic based on that information.

However, this approach will only work for some use cases. If your users are interacting with your service through a web browser, you likely want to give them a single DNS name that they can visit in the browser so that they don't need to be aware of your cells. In this scenario, creating a thin routing layer to direct the traffic becomes necessary.

The routing layer should be as tiny as possible. It should contain the bare minimum logic to identify a user (based on some information in the request), determine which cell they should be routed to, and then proxy or redirect the request accordingly.

This routing layer provides a simpler user experience (users don't need to know about your cells), but it comes at the cost of a new, global component in your architecture that you must maintain and monitor. It also becomes a single point of failure, which you can otherwise largely avoid by using a cellular architecture. This is why you should strive to keep it as small and simple as possible.

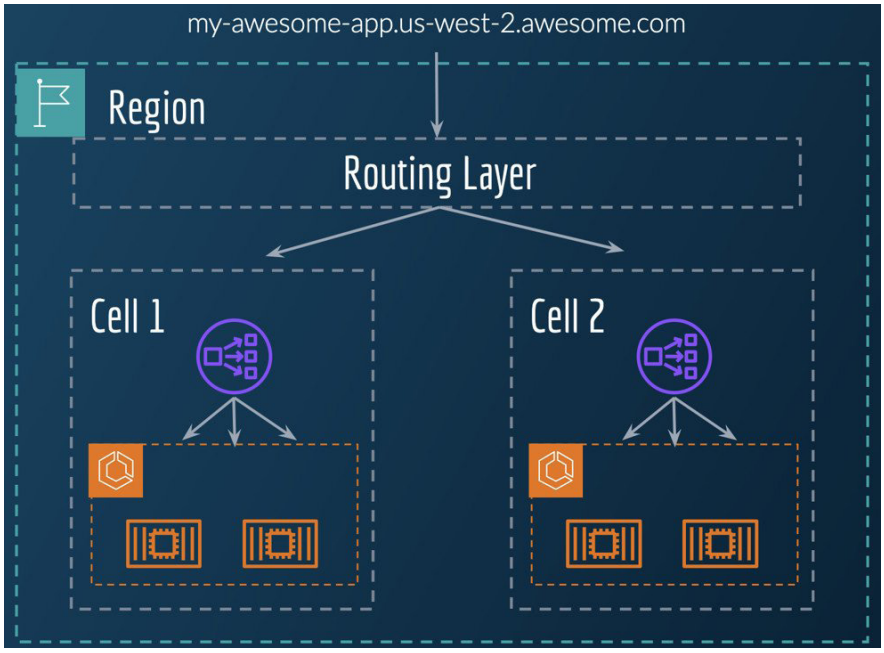


Figure 10: Routing layer for cell isolation

One nice advantage of having such a routing layer, though, is it makes it possible to transparently migrate a customer from one cell to another. Suppose a user outgrows one cell, and you would like to move them to a larger or less crowded one. In that case, you can prepare the new cell for their usage and then deploy a change to the routing logic/configuration that will redirect their traffic without them knowing anything is happening.

New Cells

If you followed along in our Standardization sections above, you have probably intuited that we've already done most of the work to solve the problem of how to create new cells. All we need to do is:

- Create a new AWS account in our Organization
- Add that account to our cell registry
- Run the cell bootstrap script to build and deploy all of the components

That's it! We have a new cell. Because we standardized the build life cycle steps in the Makefiles for each component, the deployment logic is very generalized and takes little effort to get a new cell off the ground.

Deployments

Deployments are probably the most challenging problem to solve for any application architecture, but this is especially true for cellular architecture. Thankfully, in recent years, major advancements in infrastructure-as-code tooling have made some of these challenges more approachable.

In years past, most IaC tools used a declarative configuration syntax (such as YAML or JSON) to define the resources you wished to create. However, the more recent trends have provided developers with a way to express infrastructure definition using real programming languages. Instead of grappling with complex and verbose configuration files, developers now have the opportunity to leverage the expressiveness and power of familiar programming languages to define infrastructure components. Examples of this new class of tools include:

- AWS CDK (Cloud Development Kit) - for deploying CloudFormation infrastructure
- AWS cdk8s - for deploying Kubernetes infrastructure
- CDKTF (CDK for Terraform) - for deploying infrastructure via HashiCorp Terraform

These tools let us use constructs like for loops to eliminate hundreds of lines of boilerplate YAML/JSON.

Another incredibly powerful thing about expressing our infrastructure in a language like TypeScript is that we can use npm libraries as dependencies. This means that our IaC

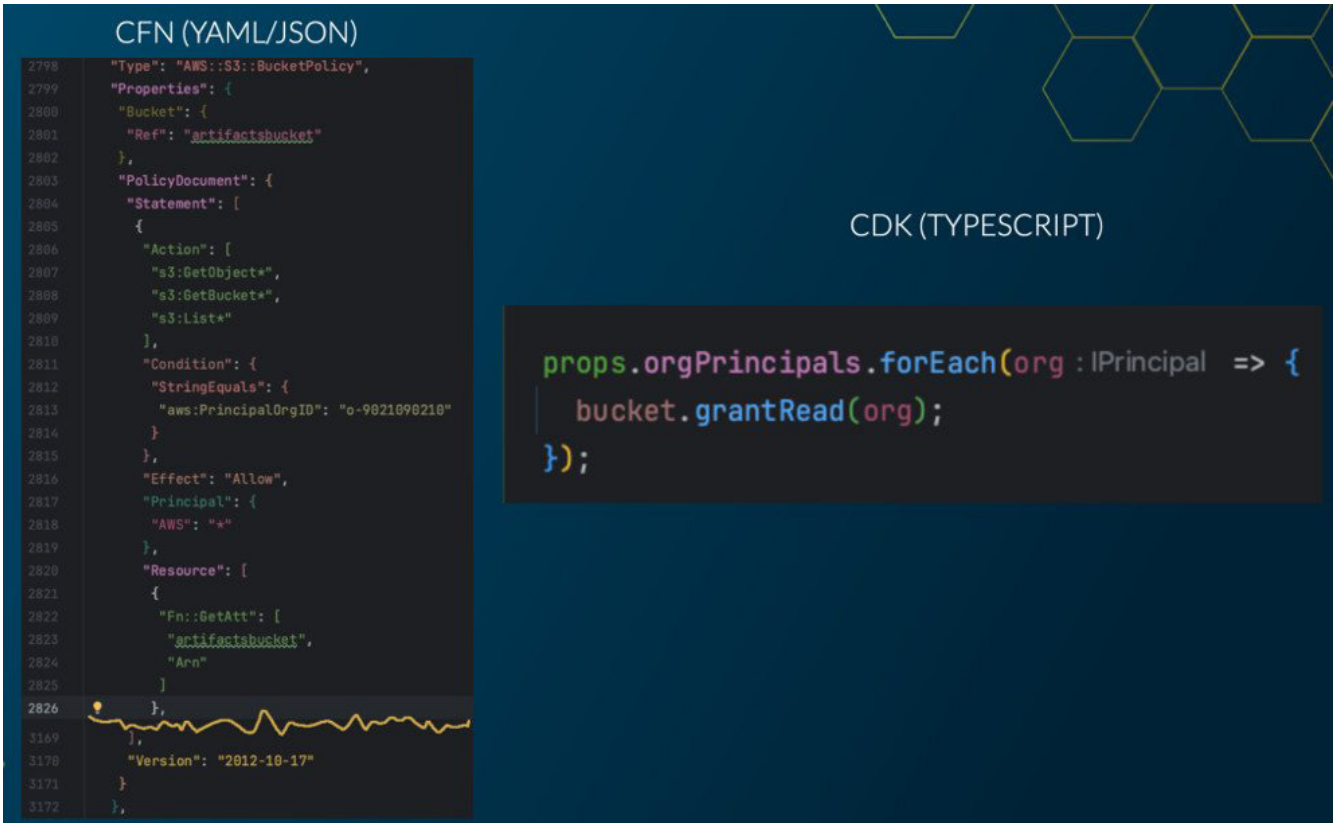


Figure 11: CloudFormation JSON vs. CDK TypeScript

projects can add a dependency on our cell registry library and thus gain access to the array containing the metadata for all of our cells. Then, we can loop over that array to define the infrastructure steps we need for each cell. When new cells are added and the cell registry is updated, the infrastructure will be automatically updated as well!

In particular, the combination of AWS CDK and AWS CodePipeline is extremely powerful. We can use a general-purpose pattern to define pipelines for each application component and set up the necessary build and deploy steps for each component while sharing most of the code.

At Momento, we have a bit of TypeScript CDK code for each type of stage that we may need to add to an AWS CodePipeline (e.g., build a project, push a docker image, deploy a CloudFormation stack, deploy a new image to a Kubernetes cluster, etc.) We can push those stages together into an array and then loop over it to add the stages to each pipeline:

We have created a special pipeline called the “pipeline of pipelines.” It is a “meta” pipeline responsible for creating individual pipelines for each application component.

This repository serves as a single source of truth for all of our deployment logic. Any time a developer needs to change anything about our deployment infrastructure, it can all be done in this one spot. Any changes we make to our list of deployment steps (such as changing the order of the cells or making the “bake” step more sophisticated) will be automatically reflected in all of our component pipelines. When a new cell is added, the pipeline-of-pipelines runs and updates all of the component pipelines to add the new cell to the list of deployment steps.

To help improve our availability story, we think carefully about what order to deploy to the production cells. Cells are

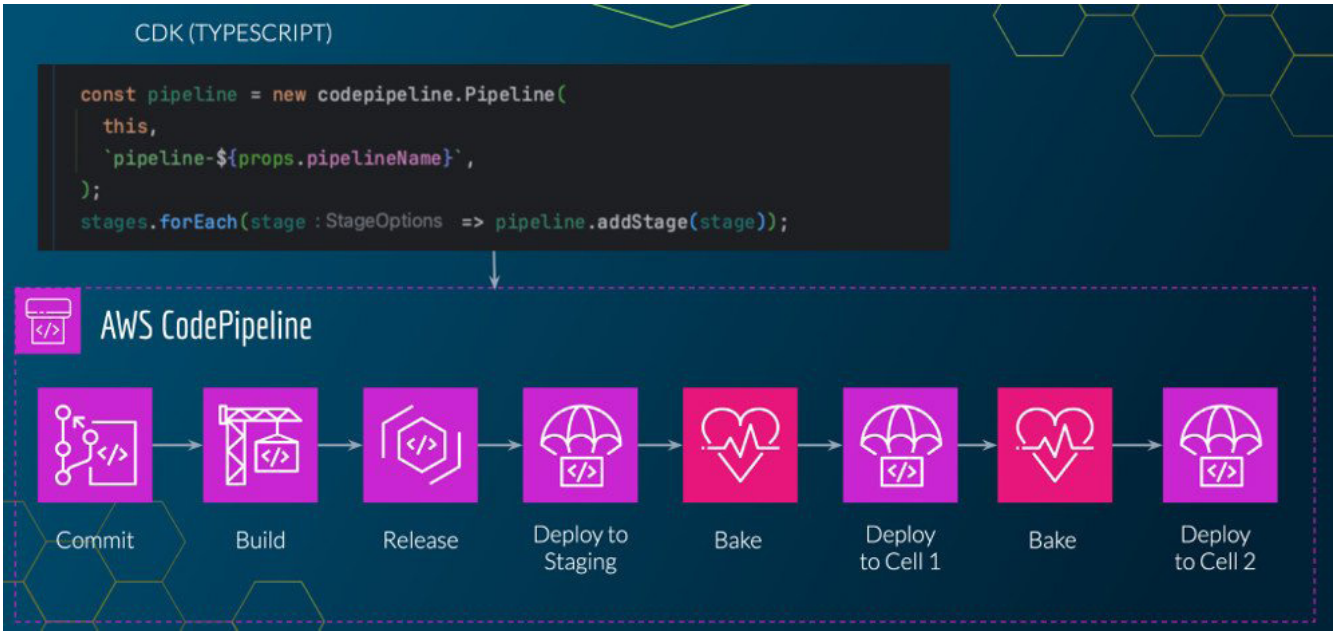


Figure 12: CDK code to add stages to a CodePipeline

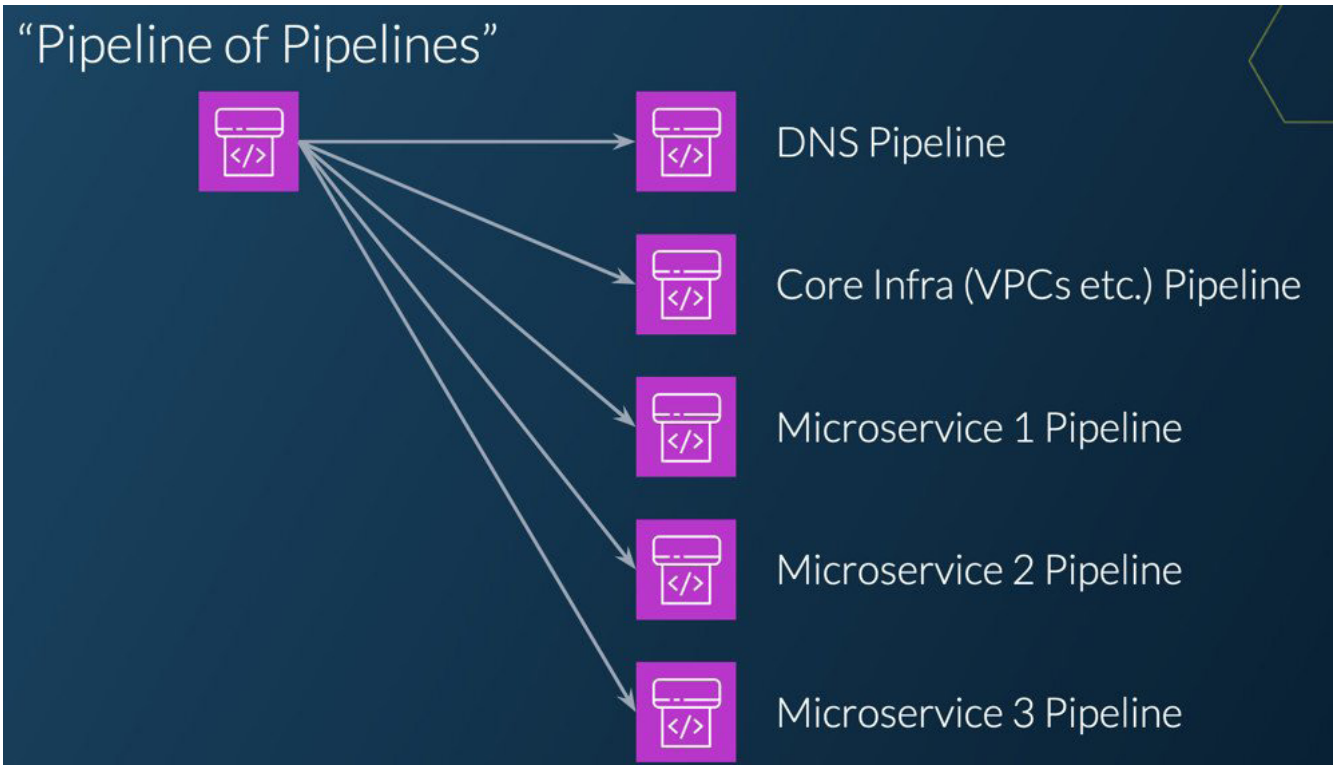


Figure 13: Pipeline of Pipelines

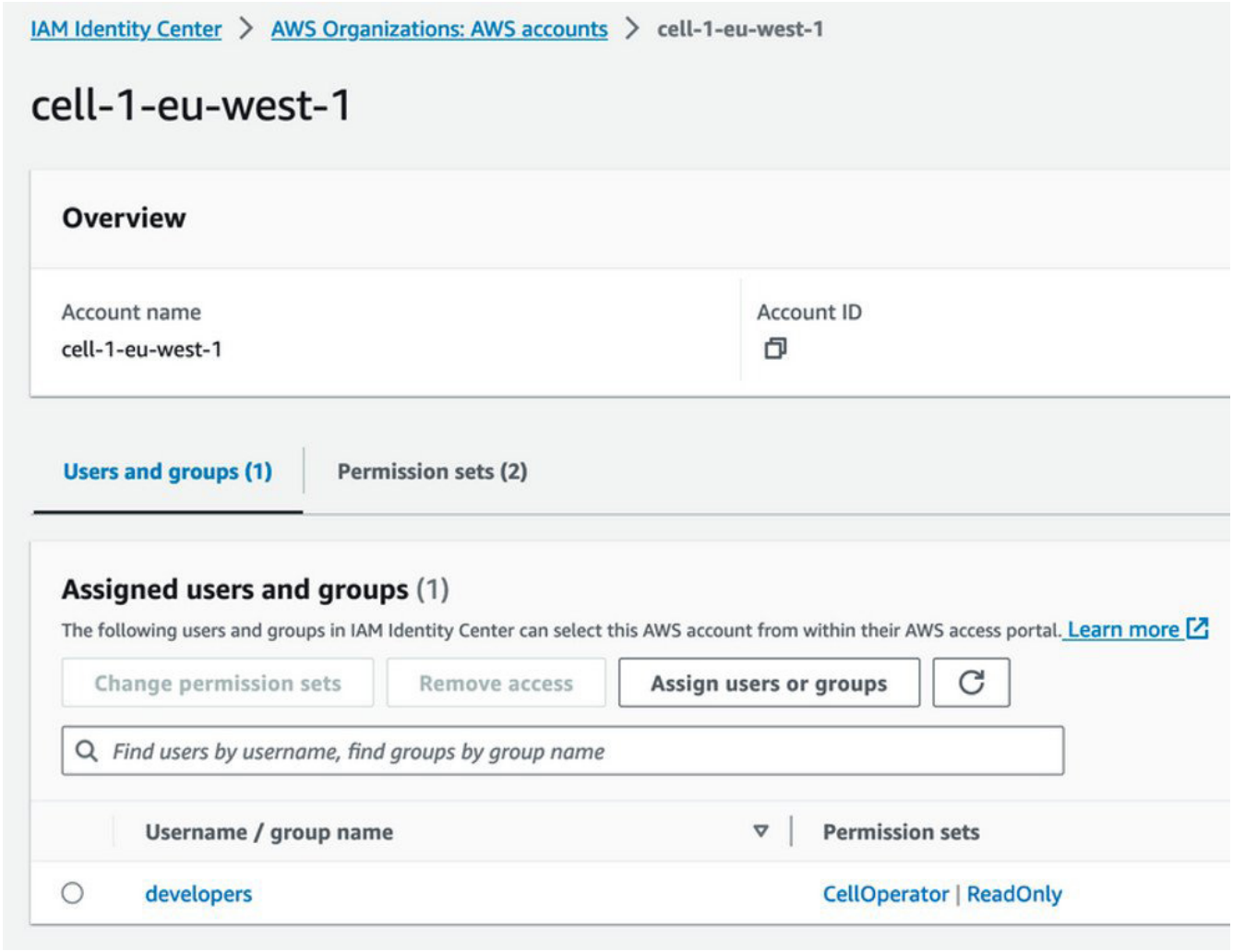


Figure 14: AWS SSO Account Permissions

organized into waves based on size, importance, and traffic levels. In the first wave, we deploy to pre-production cells, which serve as testing grounds for changes before they are promoted to production cells. If those deployments go well, we gradually deploy to larger and larger production cells. This staged deployment approach enables the controlled progression of changes and increases the likelihood that we will catch any issues before they impact many customers.

Permissions

To manage permissions into and out of the cell, we heavily rely on AWS's SSO, now known as IAM Identity Center. This service gives us a single-sign-on splash page that all of our developers can log in to using their Google identity and then access the AWS console for any accounts they have permission to access. It also provides access to targeted accounts via command-line and AWS SDKs, making it easy to automate operational tasks.

The management interface provides granular control over user access within each account. For example, roles such as "read-only" and "cell operator" are defined within a cell account, granting varying levels of permissions.

Combining the capabilities of AWS SSO's role mappings with CDK and our cell registry means we can fully automate both the inbound and outbound permissions for each cell account.

Figure 15: Metrics dashboard, with metrics grouped by cell name

For inbound permissions, we can loop over all of the developers and cell accounts in the registry and use CDK to grant the appropriate roles. When a new account is added to the cell registry, the automation automatically sets up the correct permissions. We loop over each cell in the registry for outbound permissions and grant access to resources like ECR images or private VPCs as required.

Monitoring

Monitoring a large number of cells can be difficult. It's essential to have a monitoring story that ensures your operators can evaluate the service health across all of your cells in a single view; expecting your operators to look at metrics in each cell account is not a scalable solution.

To solve this, you just need to choose a centralized metrics solution to which you can export your metrics from all of your cell accounts. The solution must also support grouping your metrics by a dimension, which in this case will be your cell name.

Many metrics solutions provide this functionality; it is possible to aggregate metrics from multiple accounts to CloudWatch metrics in a central monitoring account. Plenty of third-party options exist, such as Datadog, New Relic, LightStep, and Chronosphere.

Here is a screenshot of a LightStep dashboard where Momento's metrics are grouped by their cell dimension:

Additional Benefits

Now that we've touched on how cellular architecture helps achieve high availability and how modern IaC and infrastructure tools can help you automate your cellular infrastructure, let's note some additional benefits you can reap from this automation.

One key benefit is the ability to spin up new cells very quickly. With the cell bootstrap script described in this article, we can deploy a new cell from the ground up in hours. Without the

foundational standardization and automation, most of the steps of this process would have to be done by hand and could easily take weeks. For startups and small companies, the ability to be agile and add a new cell rapidly in response to a new customer inquiry can be a huge value proposition. It might make the difference between landing a big deal or missing out on it.

Another huge value is the ability for developers to create personal cells in their own development accounts. Sometimes, there is no real way to test and debug a complex feature that relies on interactions between multiple services or components without a real environment.

Some engineering organizations will try to solve this problem using a shared development environment, but this requires careful collaboration between developers and is prone to conflicts and downtime. Instead, with our cell bootstrap script, developers can bring up and tear down a fully operational development deployment of the application within a single day. This agile approach minimizes disruptions and maximizes productivity, allowing developers to focus on their tasks without inadvertently impacting others.

No One Size Fits All

In this article, we've discussed several tools and technologies we have chosen to automate our cellular infrastructure. But it's important to point out that for any given technology mentioned in this article, plenty of alternative choices are available. For example, while Momento uses several AWS tools, the other major cloud providers, such as GCP and Azure, offer similar products for each relevant goal.

Furthermore, you may choose to only automate a subset of the things we have chosen to automate, or you may choose to automate even more beyond what we have outlined here! The moral of the story is that you should choose the level of automation that makes sense for your business and the tools that will fit most seamlessly into your environment.

Summary

Cellular architecture can benefit your customers regarding availability and ensure you hit your SLAs. It's also valuable for your business's agility and engineering velocity. Automating these processes only requires solving a few key problems presented in this article and a little work to standardize some things across your application components.

Thanks to the changes in the infrastructure as code space, automation is somewhat simpler today, as long as you take those opportunities to standardize a few things about how you define your components.

Cell-Based Architecture Adoption Guidelines

by **Guy Coleman**, Lead Consultant @ OpenCredo

Everything fails all the time, and cell-based architecture can be a good way to accept those failures, isolate them, and keep the overall system running reliably. However, this architecture can be complex to design and implement. This article explores the best practices, problems, and adoption guidelines organizations can use to succeed.

Cell-based Architecture Best Practices

Organizations should consider several best practices when adopting cell-based architectures to improve the manageability and resilience of the systems.

Consider The Use Case

Cell-based architecture can be more complex to build and run and cost more. Not every system needs to work at the scale and reliability of S3; consider the use case and whether it's worth the additional investment. It's a good fit for systems which:

1. Need high availability.
2. Scale massively so cascading failures must be avoided.
3. Need a very low RTO (Recovery Time Objective).
4. Are so complex that automated test coverage is

insufficient to cover all test cases.

Also, consider the size of the system. For some organizations, each cell represents an entire stack: each service is deployed in each cell, and the cells don't communicate with each other (DoorDash, Slack). For others, each cell is its own bounded business context, and the system comprises multiple layers of cells communicating with each other (WSO2, Uber's DOMA). The latter may be more flexible but is undoubtedly more complex.

Make Cell Ownership Clear

If multiple cell layers communicate with each other, ideally, each cell would be owned by a single team empowered to build and deliver the cell's functionality to production.

Consider making the boundary of a cell "team-sized" to make ownership easier to establish and help the team evolve the system as required by the business. Techniques like Domain-Driven Design and Event Storming can help find these boundaries.

Isolate Cells

Cells should be isolated from each other as much as possible to minimize the blast radius for both reliability and security problems. This isn't always possible in the real world, but sharing resources should be done with care because it can significantly reduce the benefits of using cells in the first place.

On AWS, a good way to ensure isolation is to use a separate account per cell. Many accounts can be problematic to manage, but they provide excellent blast radius protection by default because you must explicitly allow cross-account access to data and resources.

It's important to consider whether a single cell should be within a single availability zone or have its services replicated in multiple availability zones to take advantage of the physical isolation that Availability Zones offer. There is a tradeoff to be made here.

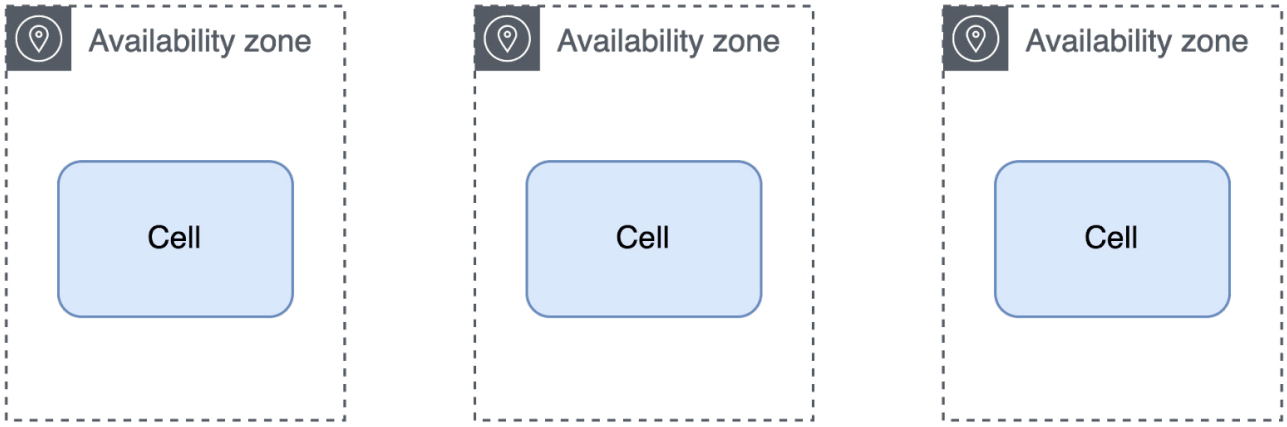
Single AZ

In a single AZ design, each cell runs in a single availability zone. (Figure 1)

The main advantage of this approach is that an AZ failure can be detected, and action can be taken to handle it, such as routing all requests to other zones.

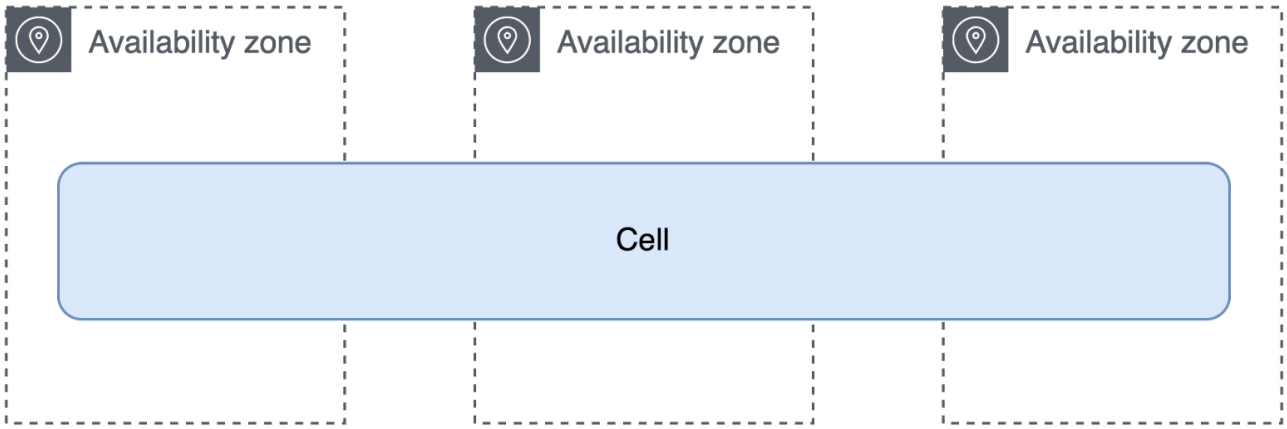
The disadvantages are:

1. Recovery can be complicated by having to replicate cell contents to another AZ, which may break the isolation properties of the cell design.
2. Depending on the router design, clients may need to be aware of the zone-specific endpoints.



Single AZ

Figure 1



Multiple AZs

Figure 2

Multiple AZs

In a multi-AZ design, each cell runs across two or more availability zones. (Figure 2)

The significant advantage of multi-AZ is using regional cloud resources (like Amazon DynamoDB) to make cells more resilient if a single zone fails.

The disadvantages are:

1. Gray failures can occur when a service experiences problems in only one AZ, making it difficult to exclude just that AZ from a given cell.
2. Also, there may be extra cross-AZ data transfer costs. DoorDash used monitoring and a service mesh with AZ-aware routing to optimize costs by keeping traffic within the same AZ where possible.

Cell Failover

What happens if the AZ becomes unavailable in a single-AZ design? Where will the affected user requests be routed to?

One answer is not to handle failover at all: cells are designed to isolate faults. The fault will have to be rectified before the affected cells return to use.

The other option is to use a disaster recovery strategy to replicate cell data to another cell in a different AZ and start routing requests to the new cell. The risk here is that replication might reduce the cells' isolation. The replication process will depend on the data requirements and underlying data stores (regional cloud services can help here: see Leverage High Availability Cloud Service).

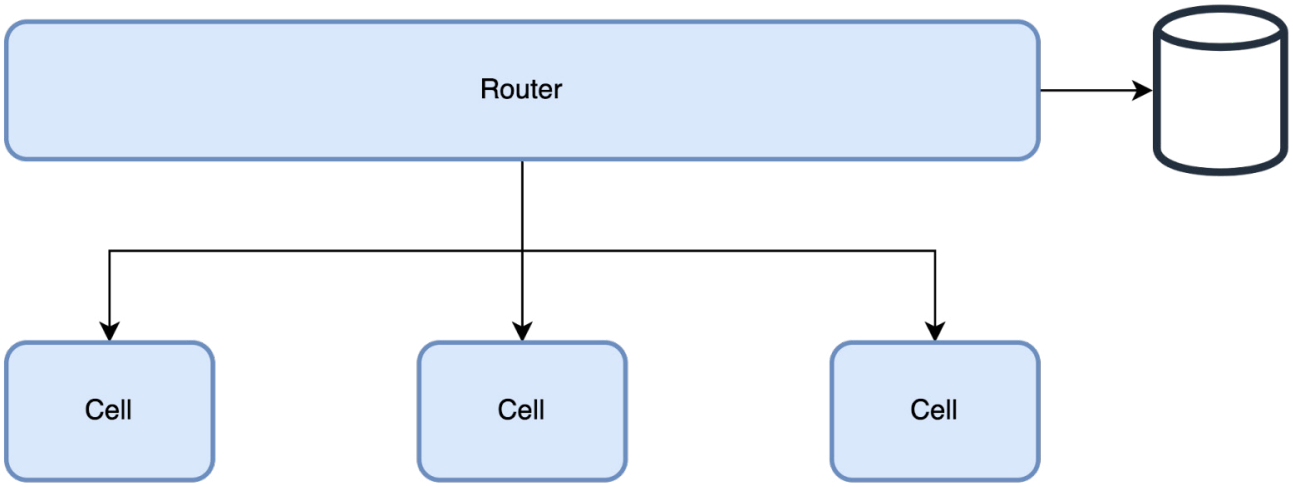
Automate Deployments

Just like with microservices, to run cells at scale, you need the ability to deploy them in hours and preferably minutes - not days. Quick deployments require a standardized, automated way of managing cells, which is vital and depends on an investment in tooling, monitoring, and processes.

Standardization does not mean that every team needs to use the same language, database, or technologies. Still, a well-understood and standard way to package and deploy applications to new or existing cells should exist. Ideally, the provisioning/deployment pipelines should allow teams to:

1. Create new cells.
2. Monitor their health.
3. Deploy updated code to them.
4. Monitor the status of a deployment.
5. Throttle and scale cells.

The deployment pipeline should reduce the complexity and cognitive load for platform users -



Router with a database

Figure 3

exactly how this looks will depend a lot on the size and tech stack of the organization.

Make Routing Reliable

The router above the cells is arguably the most critical part of the system: without it, nothing else works, and it can become a single point of failure. (Figure 3) It's essential to design it to be as simple as possible, so there are a few things to consider:

1. The technology: DNS, API Gateway, a custom service. Each has its own advantages and disadvantages (for example, managing time-to-live for DNS).
2. Leverage high-availability services. For example, if the router needs to store a customer's cell, use S3 or DynamoDB, which have very high SLAs, instead of a single MySQL instance.
3. Separate the control and data planes. For example, customer cells could be stored in S3, and the router could look up the data in the bucket. A separate control plane manages the bucket's contents, and the control plane can fail without affecting routing.
4. Think about where authentication should happen. For example, should it be:
    a. In the router, which simplifies downstream services but adds a big blast radius if it fails.
    b. In the cells, which may add complexity and repetition to each cell.
5. The router must be aware of cell locations and health to route requests away from failed or draining cells.

Limit Cell Communication

If multiple cell layers communicate with each other, it should be through well-defined APIs that help encapsulate the cell's logic and allow services inside the cell to evolve without excessively breaking the API contract. Depending on the complexity requirements, this API may be exposed directly by services in the cell or a gateway at the edge of the cell.

Avoid chatty communication between cells. Limiting dependencies between cells will help them maintain fault isolation and avoid cascading failures.

You may want to use an internal layer to orchestrate traffic between cells, such as a service

mesh, an API gateway, or a custom router. Again, care must be taken to ensure that whatever is used is not a single point of failure. Asynchronous messaging may also help as long as the messaging layer is reliable.

Leverage High Availability Cloud Service

As mentioned in the routing section above, many cloud services are already architected for high availability (often using cells like EBS and Azure AD). These services can simplify your choices and avoid reinventing the wheel.

Consider the SLAs of the cloud services, whether they are global, regional, or zonal, and how this will affect the system's performance if a given cloud service fails.

Potential Problems with Cell-based Architecture

Get Organization Buy-In

Cell-based architecture can be complex to build and run and has higher costs, so like many technology projects, it will need organizational buy-in to be successful.

For management, focusing on the business impact can be helpful, such as increased velocity (teams can more confidently deploy new code) and higher availability (happy customers and better reputation).

It will also need support and investment from architecture, DevOps, and development teams to build and run the cells with sufficient isolation, monitoring, and automation, so be sure to get them involved early to help guide the process.

Avoid Sharing Between Cells

Sharing resources like databases between cells may seem like a good way to reduce complexity and cost, but it reduces the isolation between cells and makes a failure in one cell more likely to affect other cells.

The key question is: how many cells would be affected if this shared resource failed? If the answer is many, then there is a problem, and the benefits of cell-based architecture are not being fully achieved.

A shared database can be a helpful step on a migration journey to cells but should not be shared indefinitely; there should also be a plan to split the database.

Avoid Creating an Overly-Complex Router

The router can be a single point of failure, and the risk of encountering some kind of failure increases with increased complexity. It can be tempting to add functionality to the router to simplify the cell services, but each decision must be weighed against the overall reliability of the system. Perform some failure mode analysis to identify and reduce the failure points in the router.

For example, if the router needs to look up cell mappings from a database, it may be quicker and more reliable to store the database in memory when starting up the router than relying on data access for every request.

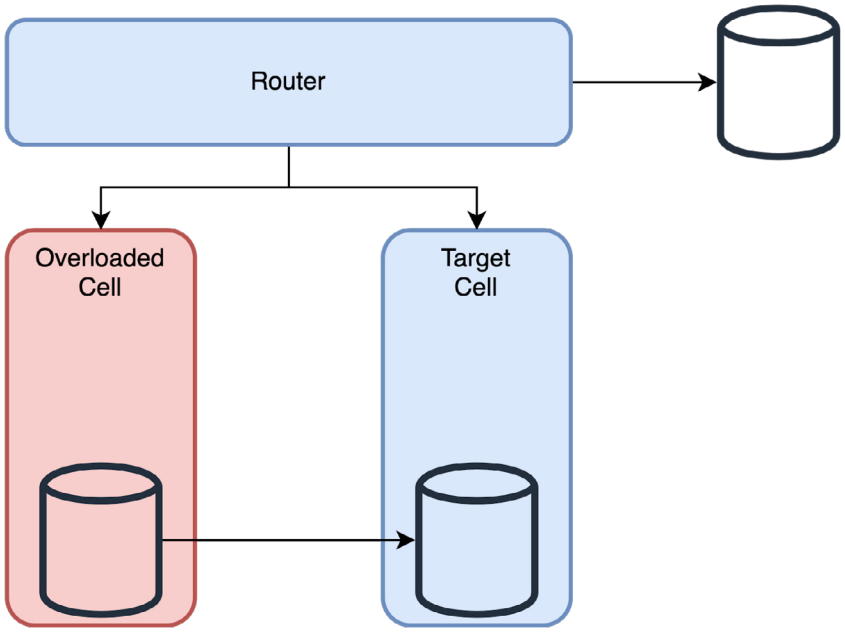
Missing Replication and Migration Between Cells

It can be tempting to consider cell migration as an advanced feature and skip it at the start of the project, but it's vital to the success of the architecture. If a cell fails or becomes overloaded (e.g. two big customers end up on the same cell), some customers need to migrate to another cell. How this looks in practice will depend on the routing and data partitioning, but the general idea is:

1. Identify a cell to migrate to (either an existing cell with capacity or a newly created one).
2. Replicate any required data from the old cell's databases into the target one.
3. Update the router configuration to make the target cell active for the relevant customers.

Integration with the routing layer is also needed to ensure requests are routed to the right cell at the right time.

Replication may be triggered by cell failure, or cells may be replicated so another cell is always ready to go. Exactly how this replication looks will depend on the cell's data schema, recovery point objective (RPO), and recovery point objective (RTO) requirements: database-level replication, messaging, and S3 are all options. See the Disaster Recovery of Workloads on AWS whitepaper for more discussion of recovery strategies.



Migrating an overloaded cell

Figure 4

Avoid Limits on Cloud Resource

If the system consumes many cloud resources per cell, it may hit soft or hard limits imposed by the cloud provider. It may be possible to request increases to soft limits, but hard limits can be imposed by service or hardware constraints and are fixed.

On AWS, many limits can be avoided by using a separate account per cell.

Balance Duplication of Logic and Data

There is a tradeoff between keeping cells as isolated as possible and avoiding duplication of logic and data between services. The same tradeoff exists with microservices and the "Don't Repeat Yourself" (DRY) principle.

As systems grow, it can be better to avoid tight coupling and promote isolation by duplicating code between services in different cells and potentially even duplicating data if it makes sense. There is no generic right or wrong answer to this problem: it should be assessed on a case-by-case basis. Conducting failure mode analysis can help identify when a dependency between cells might be a problem and when it should be removed, possibly by duplication.

Adoption Guidelines

You've decided a cell-based architecture is a good fit - now what?

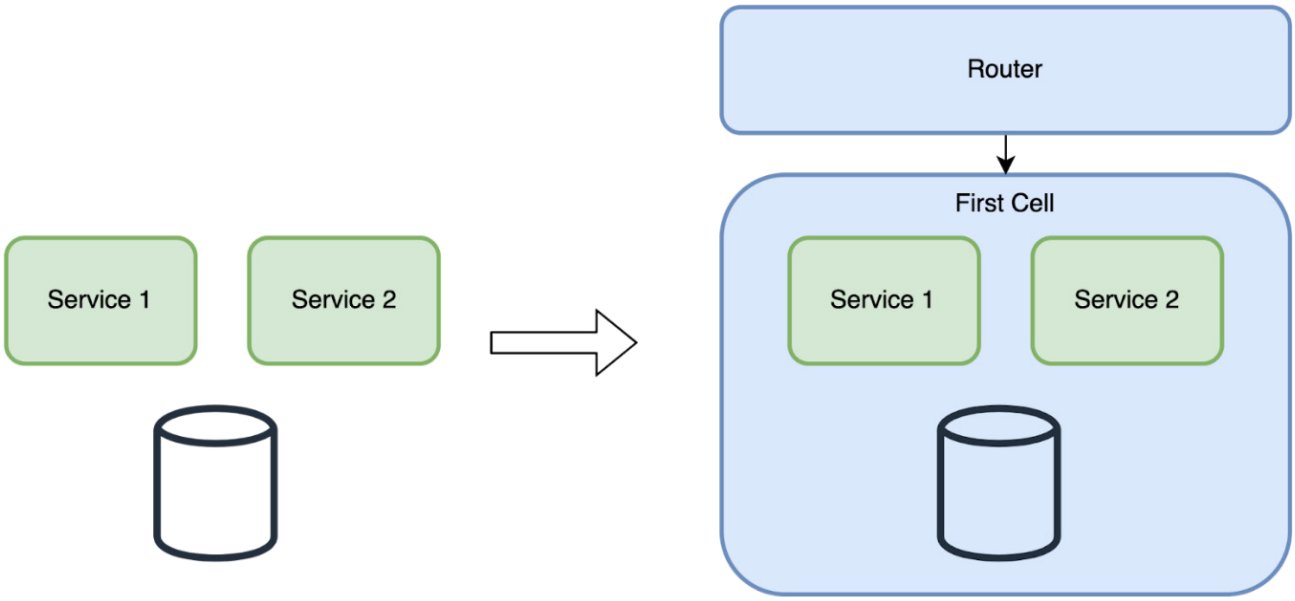
Migration

To quote Martin Fowler: if you do a big-bang rewrite, the only thing you're certain of is a big bang.

Migrating an existing microservice architecture to become cell-based can be tricky. A common first step is to define the first cell as the existing system with a router put on top and then peel off services into new cells; in the same way, a monolith-to-microservice migration might happen. (Figure 5)

Organizations can use many monolith-to-microservice strategies. For example:

1. Use Domain-Driven Design (DDD) to define bounded contexts and help decide what goes in new cells.
2. Migrate service logic into separate cells first, then split shared data into cell-specific databases in a subsequent phase.



An existing system becomes the first cell

Figure 5

3. Consider which business areas would benefit from greater resiliency when choosing what to split into cells first.
4. Ensure sufficient automation and observability to manage the new, more complex system.

Deployment

In a cell-based architecture, the unit of deployment is a cell. New application versions should be deployed to a single cell first to test how they interact with the rest of the system while minimizing the risk of widespread breakage. Use techniques like canaries or blue/green deployments to make incremental changes and verify that the system is still performing as expected before continuing the rollout (usually in waves).

If the new version has problems, the changes should be rolled back, and the deployment should be paused until further investigation can pinpoint the issue.

The concept of 'bake time' is also crucial to ensure the new cell has enough time to serve real traffic for monitoring to detect problems. The exact time will vary - minutes or hours depending on the kind of system, risk appetite, and complexity.

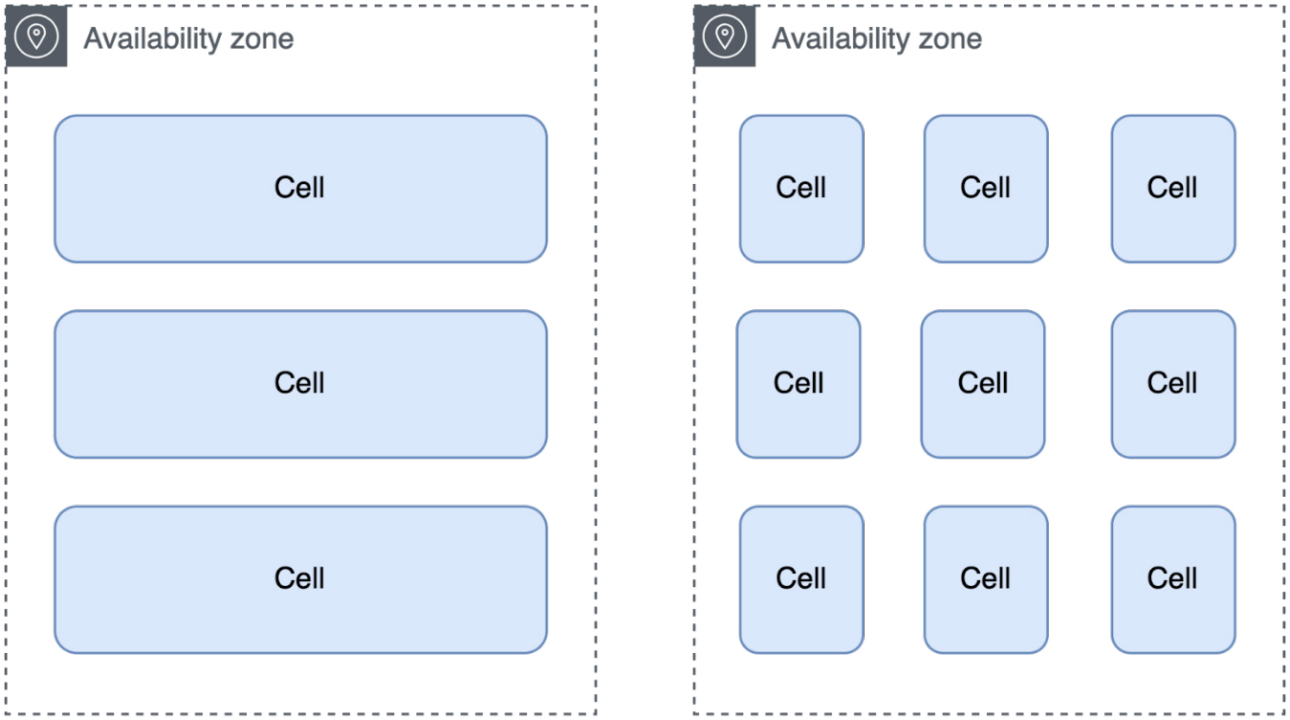
Observability

In addition to monitoring microservices the right way, there should be additional cell monitoring and dashboards to see aggregate and cell-level views of:

1. The number of cells.
2. Cell health.
3. Status of deployment waves.
4. Any SLO metrics important for a cell.

Many of these can be derived from standard cloud metrics, but additional tagging standards may be needed to get cell-level views.

Because cell-based architecture risks increasing cloud usage and, therefore, cost, it's essential to keep track of resource usage and cost per cell. The goal is to allow teams to ask questions like "How much does my cell cost?", "How



Larger cells vs smaller cells

Figure 6

can I make more efficient use of the resources?" and "Is the cell size optimized?".

Scaling

In a cell-based architecture, the unit of scaling is a cell: more can be deployed horizontally in response to load. The exact scaling criteria will depend on the workload but could include the number of requests, resource usage, customer size, etc. How far scaling can be taken will depend on how isolated the cells are—any shared resources will limit scalability.

The architecture should also be careful to know the limits of a cell and avoid sending it more traffic than its resources can handle, for example, by load shedding by the router or cell itself.

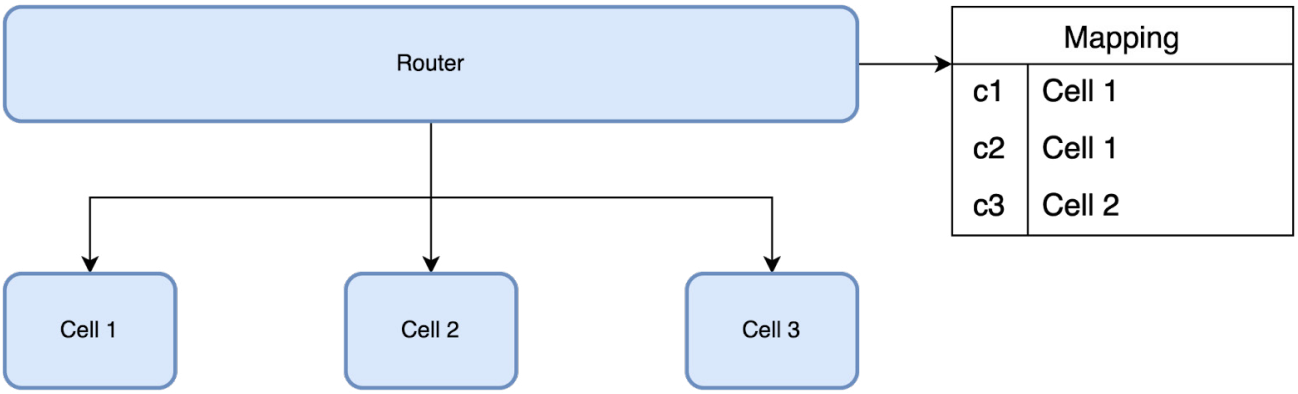
Cell Sizing

Deciding on the size of each cell is a crucial tradeoff. Many smaller cells mean a smaller blast radius because each cell handles fewer user requests. A small cell can also be easier to test and manage (for example, a quicker deployment time).

On the other hand, larger cells may make better use of the available capacity, make it easier to fit a large customer into a single cell, and make the whole system easier to manage because there are fewer cells. (Figure 6)

It's a good idea to think about:

1. The blast radius.
2. Performance. How much traffic can be fitted into a cell, and how does it affect its performance?
3. Headroom in case existing cells need to start handling traffic from a failed cell.
4. Balancing the allocated resources to ensure cells are not underpowered to handle their expected load, but not overpowered and costing too much.



Database of cells per customer

Figure 7

The advantages of smaller cells are:

1. They have a smaller blast radius so that any failure will affect a smaller percentage of users.
2. They are less likely to hit any cloud provider quota limits.
3. They lower the risk of testing new deployments because targeting a smaller set of users is easier.
4. Fewer users per cell mean migrations and failovers can be quicker.

The advantages of larger cells are:

1. They are easier to operate and replicate because there are fewer of them.
2. They utilize capacity more efficiently.
3. Reduce the risk of having to split large users across multiple cells,

The correct choice will depend heavily on the exact system being built. Many organizations start with larger cells and move to smaller ones as confidence and tooling improve.

Data Partitioning

Closely related to cell sizing is partitioning data and deciding which cell customer traffic should be routed to. Many factors can inform the partitioning approach, including business requirements, the cardinality of data attributes, and the maximum size of a cell.

The partition key could be a customer ID if the requests can be split up into distinct customers. Each cell is assigned a percentage of the customers so that the same customer is always served by the same cell. If some customers are larger than others, then care should be taken that no single customer is bigger than the maximum size of the cell.

Other options are geographical region, market type, round-robin, or load-based.

Whatever approach is used, it can also be beneficial to override the router and manually place a customer in a specific cell for testing and isolating certain workloads.

Mapping

Using a customer ID implies the router will need to map customers to cells. The most straightforward approach to storing the mapping data could be a table that maps every customer to a cell. (Figure 7)

The significant advantage is it's pretty simple to implement and simplifies migrating customers between cells: just update the mapping in the database.

This approach's disadvantage is that it requires a database, which may be a single point of failure and cause performance concerns.

Other approaches are consistent hashing and mapping a range of keys to a cell. However, they are both less flexible as they risk hot cells, making migrations more challenging.

Measure Success

Ideally, organizations should consider adopting cell-based architecture to achieve specific business goals, such as improving customer satisfaction by improving the technology platform's stability.

Through migration, it should be possible to measure what progress is made towards those goals. Often, the goal is resiliency in the face of failure, where some quantitative measures are useful:

1. Health metrics, including error rates or uptime (e.g., when EBS moved to cells, the error rate dropped dramatically).
2. MTTR (mean time to repair).
3. Performance metrics, including p75, p95, and p99, request processing times to see if the extra layers adversely impact latency. Performance may improve if customers are now served from smaller cells than the previous system!
4. Resource usage to make sure that cost is not getting out of control and can be optimized if necessary.

These all imply good observability to measure performance, reliability and cost.

Conclusions

Cell-based architecture can be daunting and complex to implement, but many good practices will be familiar to microservice developers. Any architecture at this scale should include deployment automation, observability, scaling, and fault recovery; cell-based architecture is no exception. These must be considered when designing cell size, cell isolation, data ownership, and a strategy to recover from failures.

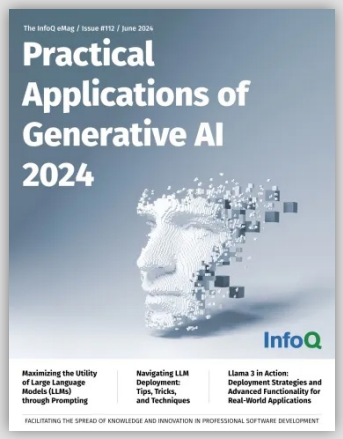
Perhaps the crucial decision to be made is around data partitioning and, closely related, how request traffic is allocated and mapped to cells. More straightforward approaches may be easier to implement, but they typically lack the flexibility required to run cells at scale.

The public cloud providers offer many high-availability services that can be leveraged to improve reliability while simultaneously simplifying the design. AWS has the most presence online for cell-based architecture, with talks about how they applied this pattern to their own systems and advice for using AWS services to implement it.

Organizations must ensure that the cell-based architecture is the right fit for them and that the migration will not cause more problems than it solves. Migrating an existing system to a cell-based architecture can be done in steps to minimize disruption and validate the changes work as expected before proceeding.

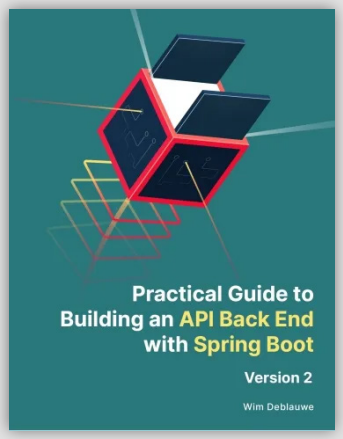
The challenges in building modern, reliable, and understandable distributed systems continue to grow, and cell-based architecture is a valuable way to accept, isolate, and stay reliable in the face of failures.

Curious about previous issues?



Generative AI (GenAI) has become a major component of the artificial intelligence (AI) and machine learning (ML) industry. In the InfoQ “Practical Applications of Generative AI” eMag, we present real-world solutions and hands-on practices from leading GenAI practitioners.

Learn from Netflix, Pinterest, AWS, Cloudflare how to foster scalable and highly available architectures. These organizations cover diverse business scenarios, all of which push the conventional boundaries of software architectures with metrics and challenges seldom encountered in smaller companies.



If you are eager to learn about Spring Boot, REST APIs, security, validation, and testing, consider giving this updated book a try. You are guaranteed to learn something new!

InfoQ