
O'REILLY®
Compliments of
kyndryl
Cloud Adoption for Mission-Critical Workloads
Principles for Designing Always On Applications
Haytham Elkhoja
REPORT


kyndryl
The Heart of Progress™
Modernizing and managing the world's mission-critical systems
kyndryl.com

Cloud Adoption for Mission-Critical Workloads

Principles for Designing Always On Applications

Haytham Elkhoja

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Cloud Adoption for Mission-Critical Workloads
by Haytham Elkhoja

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Megan Laddusaw
Development Editor: Melissa Potter
Production Editor: Jonathon Owen
Copyeditor: nSight, Inc.

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Kate Dullea

June 2023: First Edition

Revision History for the First Edition
2023-06-13: First Release

This work is part of a collaboration between O'Reilly and Kyndryl. See our statement of editorial independence.

978-1-098-14949-9

[LSI]

Table of Contents

Introduction

Mission-critical applications are systems that play a fundamental role in the operations of an organization. They are often used to perform core business processes such as financial transactions, traffic control coordination, and emergency response.

Such applications are essential, and their failure or unavailability can have serious consequences for the business or the well-being of the end user. These applications must be built to guarantee that they are reliable, dependable, and able to perform their functions and return satisfactory results in a timely manner—even in the most challenging of circumstances.

Cloud adoption has been increasingly important for organizations, as it gives them access to a comprehensive catalog of on-demand and scalable services, platforms, and infrastructures without having to invest in hardware and software. The cloud offers many advantages to help accelerate time to value and innovate in a more flexible, agile, and efficient way (which, in turn, helps reduce costs, increase operational efficiency, and enable businesses to respond quickly to changing market conditions). Cloud providers like Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure are revolutionizing this adoption.

Despite the fact that the majority of organizations have embraced these platforms in one way or the other, a McKinsey & Company[1] study shows that only 20% of workloads have migrated (or have

1 Arul Elumalai et al., "IT as a Service: From Build to Consume," McKinsey & Company, September 15, 2016.

modernized) to the cloud. This is partly because enterprises that manage sensitive and mission-critical workloads are conscious of the challenges and complexities when it comes to adopting the cloud.

This report explores ways to reconcile the potential of cloud platforms with the intricate requirements of reliability for mission-critical workloads. It provides strategies for the successful adoption of cloud platforms and cloud-native patterns while simplifying the end-to-end operational complexity of such workloads.

CHAPTER 1

Drivers and Considerations

End users have come to have certain expectations in terms of the consistency and reliability of mission-critical services. Downtime, whether planned or unplanned, can cause significant disruptions to an organization's operations, and even a few minutes of unavailability can result in substantial loss of revenue, impact the reputation of the organization, decrease customer loyalty, and even potentially affect the livelihood of end users.

Depending on the industry, the impact of such outages can vary significantly. For instance, a few minutes of downtime for an online retailer could cost them hundreds of thousands of dollars in missed sales and, more importantly, erode customer trust—but the safety and well-being of patients could be seriously impacted by a comparable outage for a healthcare organization.

Failures, Outages, and Disasters

Unplanned outages, typically caused by unexpected hardware or software failures, cause significant disruptions and impact the business. Additionally, planned outages that are conducted for maintenance or application upgrades can also result in service unavailability that impacts the overall user experience, the immediate response, and the consistent operation of the service that users come to expect.

Outages and their criticality differ. Table 1-1 shows some of the more important ones.

Table 1-1. Outage examples

Types of outages	Examples
Human	New code, misconfigurations, faulty deployments, mistyping commands, and untested changes to production systems
Software	Bugs in the code, unexpected interactions between components, and security vulnerabilities
Distributed	Timeout errors, lost messages, duplicate messages, delayed messages, incomplete messages, and out-of-order messages; degradation of external third-party APIs
Capacity	Noisy neighbors and poorly sized systems and components that become overwhelmed or unable to handle the demand placed on them
Networking	Erroneous DNS, WAN, BGP, and other types of routing configurations and latency timeouts
Data	Data loss, data rots, corruptions, accidental deletes, and poorly orchestrated database schema changes
Natural	Earthquakes, storms, floods, heatwaves, and other disasters that can disrupt entire regions
Cybersecurity	Distributed denial-of-service (DDoS), malware infections, ransomware attacks, and expired certificates
Energy	Power distribution; mechanical and HVAC disruptions

All these types of outages are problematic because they can quickly cascade and escalate from a small issue to a major crisis, making it difficult for operation teams to respond to and recover from the failure. What's more, although it's not always acknowledged, enterprises find that the underlying reason for failures is often simply unknown.[1]

The Uptime Institute's 2022 Outage Analysis report[2] indicated that high outage rates haven't changed significantly even with an increase in cloud adoption and that downtime costs are actually rising: over 60% of outages cost more than $100,000, an increase from 39% in 2019, and 15% of outages cost more than $1 million, an increase from 11% in 2019.

1 Haryadi S. Gunami et al., "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages," SoCC '16, October 2016.

2 "Uptime Institute's 2022 Outage Analysis Finds Downtime Costs and Consequences Worsening as Industry Efforts to Curb Outage Frequency Fall Short," Uptime Institute, June 8, 2022.

Despite efforts to create resilient applications hosted on redundant systems and supported by mechanisms that enable services to "recover" or "failover," recovery processes are long, tedious, and often fraught with risk, and they result in a subpar experience for both the organization and its users.

Faced with this reality, many organizations running critical services are cognizant that they can't afford the time to recover from failures and that their online services should be continuously available, regardless of the situation and circumstances.

Resiliency Versus Reliability

Resiliency and reliability are both important concepts in IT, and it could be argued that resiliency is a subset of reliability. It is, however, important to define them in order to contextualize their different characteristics:

Resiliency
: *Resiliency* refers to the ability of a system, process, or organization to quickly recover from disruptions or failures. Fundamentally, resiliency is how swiftly we can bring the operation of a service back to its normal state.

Reliability
: *Reliability*, on the other hand, refers to the dependability of a system, process, or organization and its ability to consistently perform its intended function. In essence, reliability refers to the ability of an online service to perform its intended function consistently and without interruption. Reliability is closely related to robustness—specifically, a system that exhibits robustness is more likely to be reliable, as it can provide consistent behavior-facing variations.

Another concept to consider is cyber resilience, which is pivotal in today's digital landscape, as it enables organizations to quickly detect, respond to, and recover from cyber threats and attacks. While its importance cannot be overstated, discussing cyber resilience is not within the scope of this report.

In today's always-connected world, end users expect reliable, stable, and seamless access to services—that is, the significance of reliability surpasses that of resiliency. I call this *Always On*.

Always On Mission-Critical Services

Always On refers to the ability to withstand component and application failures, catastrophic events, and the introduction of updates and changes in a nondisruptive and transparent manner. Making applications Always On (or reliable) is rapidly becoming a strategic goal at the highest levels of organizations that seek to achieve it for their own reasons:

Customer satisfaction
: An Always On service ensures that customers can access business services at any time, leading to greater satisfaction and loyalty.

Revenue generation
: An Always On service is critical to generating revenue since any downtime can result in lost sales and a damaged reputation.

Risk management
: An Always On service can help mitigate risks associated with cyberattacks, system failures, or other emergencies that can impact business operations.

The reality is that many mission-critical applications typically consist of hundreds of dependencies and distributed systems, some of which are more complex than others. The challenge is that when mission-critical workloads are modernized to the cloud, it is also necessary to adapt and modernize how to meet their demanding nonfunctional requirements, such as reliability, scalability, and observability, as well as those of their dependencies.

The traditional reliance on infrastructure and platforms to address these nonfunctional requirements is no longer the method of building and deploying sensitive workloads. It has been demonstrated, often publicly,[3] that a cloud platform's availability does not necessarily imply an application's availability.

This is precisely why enterprises are still hesitant to place complete trust in public cloud companies with critical workloads. The Uptime Institute found that 32% of respondents would only trust certain

3 John Graham-Cumming, "Cloudflare Outage on July 17, 2020," *Cloudflare* (blog), July 17, 2020.

workloads with public clouds, while 14% do not trust the cloud with critical loads at all.[4]

One of the reasons behind that lack of trust stems from failed past cloud migrations that often involved a straightforward lift-and-shift model from on-premises infrastructure to a "resilient cloud," considering this as sufficient. ISG reports that over 60% of enterprises use lift-and-shift as one of their preferred approaches[5] to move applications to the cloud.

Often, the root cause for unsuccessful cloud adoptions is the misconception that architectures and deployment models used on-premises and on-cloud are analogous. On-premises resiliency is traditionally implemented through redundant infrastructure components, manual configuration, complex recovery runbooks, and, sometimes, fragile clustering intelligence. Cloud and cloud-native reliability, in contrast, must be implemented through application-level reliability patterns that are inherent to the cloud and to modern software development frameworks and patterns.

This shift toward making the application equally responsible for reliability instead of solely depending on the infrastructure's resiliency represents a major change in perspective. And while cloud providers offer building blocks and capabilities to help architects, developers, and site reliability engineers address these needs, a holistic approach is required that considers all aspects of the system.

What's more, the system should be supported by an institutional culture that prioritizes reliability at all levels of the technology stack. This leads to an end-to-end design of the services that must consider the application, data, infrastructure, deployment, people, and culture to effectively face all the challenges of building and running mission-critical cloud services.

In the next chapter, I delve into the importance of considering reliability in every aspect of the cloud application architecture, deployment method, and operation to achieve Always On.

4 Andy Lawrence, "Is Concern Over Cloud and Third-Party Outages Increasing?," Uptime Institute, June 8, 2022.

5 Stanton Jones, "Index Insider: Cloud Is a Forcing Function for ADM," *ISG* (blog), 2023.

CHAPTER 2

The Always On Strategy

Simply put, Always On mission-critical services are applications that have been purposefully designed and deployed to fulfill the demanding requirement of zero tolerance for recovery time (i.e., applications that cannot afford the time needed to recover).

Contrary to other resiliency methods presented in Figure 2-1, such as disaster recovery and warm/standby, Always On aims to provide zero downtime as perceived and experienced by the end user. This means that the goal of achieving a zero downtime strategy is not simply a technical issue but a customer-centric one.

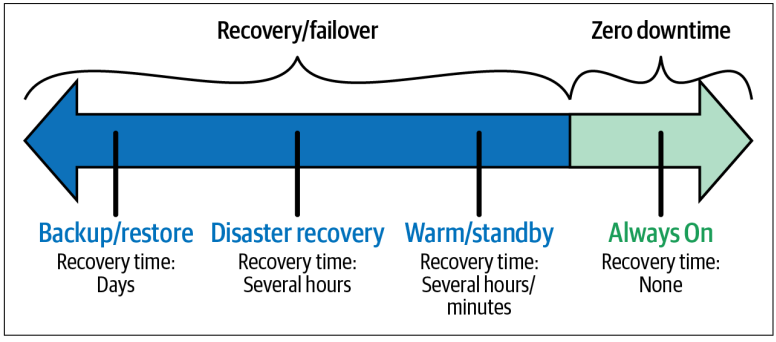

Figure 2-1. Recovery and zero downtime strategies

Cloud SLAs Are Often Misunderstood

Cloud service-level agreements (SLAs) address only a few aspects of the overall user experience. The confusion stems from the limited

scope of SLAs provided by cloud providers, which usually only apply to the infrastructure and managed services provided "as a service." And that simply isn't enough. For example, and despite being designed with high availability and fault tolerance in mind, cloud Kubernetes platforms often experience failures, and enterprises have little to no control over how quickly the service will recover.

Reliability is impacted by multiple layers that all play a role in delivering a service. It is essential to understand the varying impact each layer has on the overall customer experience since this is the only metric that truly matters to the customer. The novel distinction here is that when adopting an Always On strategy, organizations must be concerned about the availability levels of the end-to-end business service rather than solely on the availability of the underlying cloud infrastructure and platform. This mindset shift has been perceived as a potentially expensive endeavor by many organizations, which is why it is important to measure and assess the cost of interrupted services in contrast to the cost of ensuring continuous availability.

A simplistic way to measure this cost is to consider its impact on revenue and reputation. For instance, an outage in the booking system of a major airline would not only affect direct sales and revenue but would also attract negative publicity. On the other hand, an outage to its critical internal backend systems that results in grounding planes and disrupting hundreds of thousands of passengers would result in flight cancellations, compensation to passengers, damage to the airline's reputation, regulatory measures, and potential suspension of the supply chain necessary for the smooth operation of the airline.

To that end, when undertaking an Always On transformation while adopting the cloud, enterprises must establish a shared consensus on the desired level of the reliability they aim to provide to their customers and stakeholders. By doing so, they offer a clear framework for measuring and improving service over time.

To measure the reliability of their business services, such as booking a flight ticket or completing an ecommerce transaction, a customer-centric service-level objective (SLO) must be agreed upon by all parties involved. This will help provide a measurable user experience metric and quantify the potential number of transactions lost during an outage, which can be translated into monetary terms.

Service-Level Objectives Versus Recovery-Time Objectives

An SLO is a documented target for the level of service that a system aims to provide to its consumers. It is often used as a measurable goal that specifies a desired level of performance for a particular subsystem, service, or component. Examples of SLO metrics include uptime, API response time, and error rates.

On the other hand, a recovery-time objective (RTO), a metric used in business continuity and IT disaster recovery strategies, is the maximum amount of time it should take to restore a system or service to normal operations after a disruption. RTO is an important metric to measure, but it does not consider the time (often lengthy) it takes for an organization to detect, agree on, and declare an outage. RTOs also don't reflect whether recovery systems and data centers have adequate operational capacity and performance to match production.

RTOs and SLOs are related concepts, but they serve different purposes. This report recognizes RTO as a critical component in achieving a comprehensive SLO.

End-to-End Service-Level Objectives

While SLOs and other reliability parameters such as MTBF (mean time between failures), MTTR (mean time to repair/recover/resolve/respond), and MTTF (mean time to failure) are well established in incident management, they are used to measure local resource failures but do not trace a global end-to-end user experience. In an Always On approach, organizations should focus instead on personalized and customer-centric (i.e., following a customer's journey through an online service), transactional-level SLOs known as end-to-end SLOs.

An end-to-end SLO measures the level of service delivered to the end user, from the end user's point of view, through the entire service delivery chain. It takes into account the transactional critical path and the entire technology stack involved in delivering the service, including the user's browser, internet transport, application, networking, compute, messaging, APIs, runtimes, third-party dependencies, and other infrastructure components. In essence,

end-to-end SLOs are composite (or nested) SLOs, meaning that they are created by combining multiple local SLOs, allowing for a holistic view of the business service and the customer's experience.

End-to-end SLOs are expressed in terms of the percentage that a business service is operational and responding properly and accurately to end users over a given period of time. This percentage is commonly referred to as *nines* and is used to indicate the level of reliability that is expected for a mission-critical service. The more nines a service has during a month, the more reliable and available it is. The level of nines required for a particular service will depend on the specific needs of the business service and the criticality of the application. The end-to-end SLOs that address reliability are shown in Table 2-1.

Table 2-1. Service levels

End-to-end SLO	Outage per year	Outage per month	Mission-critical compatibility
99%	3.65 days	7.20 hours	This level of reliability is not considered acceptable for mission-critical business services.
99.9%	8.76 hours	43.2 minutes	This level of reliability is not considered acceptable for mission-critical business services.
99.99%	52.6 minutes	4.32 minutes	This level of reliability is considered good enough and is often used as a target for mission-critical services.
99.999%	5.26 minutes	25.9 seconds	This level of reliability is considered the standard of excellence for mission-critical services and is what we consider to be Always On.

Calculating service levels is beyond the scope of this report; however, many[1] resources[2] discuss these in great length.

1 Cat Chu and Gang Chen, "Composite Availability: Calculating the Overall Availability of Cloud Infrastructure," *Google Cloud* (blog), November 15, 2022.

2 "Using Business Metrics to Design Resilient Azure Applications," Microsoft Azure, November 30, 2022.

While end-to-end SLOs can be valuable to guide organizations in the development, architecture, deployment, and operation of mission-critical workloads, organizations must be careful to acknowledge their nuances. The frequency and impact of service disruptions can have varying degrees of importance for an organization, and most metrics don't recognize the difference.

For instance, an end-to-end SLO may not differentiate between a single outage lasting five hours and five outages lasting one hour each. While both present the same availability metric (in terms of nines), they differ in reliability. Another aspect to consider is that end-to-end SLOs may not account for variations such as peak hours versus periods of lower activity.

End-to-end SLOs establish a beacon of guidance and a shared goal within the enterprise. This can foster greater collaboration and communication between business stakeholders, engineering, and IT, enabling them to use a new architectural approach and speak a common language. This is why ownership, roles, and responsibilities must evolve and involve the entire enterprise to ensure that funding is available and that culture will adapt for its implementation.

Achieving 99.999% End-to-End SLOs in the Cloud

It is crucial then to have ongoing conversations at the highest levels of the organization to understand and recognize the importance of reliability during a transformation and modernization to cloud providers. One of the differences in designing for reliability between on-premises and cloud providers lies in the fact that Always On in the cloud replaces high availability and disaster recovery as a single, comprehensive, and unified strategy, rather than treating them separately (see Figure 2-2).

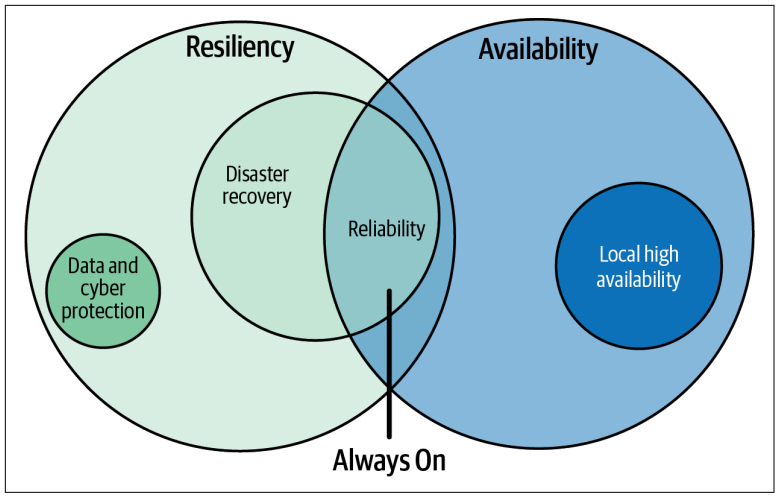

Figure 2-2. Always On: availability and reliability

Cloud providers offer distributed services around the globe and provide multiple building blocks of reliability, resiliency, and redundancy. By leveraging these capabilities, Always On workloads can be deployed across multiple active geographically distributed cloud regions and isolated fault domains, which eliminates the need for separate high availability and disaster recovery solutions.

Moving an application to the cloud as is, however, does not necessarily guarantee its reliability. It is therefore important to design it, develop it, and deploy it in a way that considers the unique features and capabilities of cloud providers while doing so in a cloud-agnostic manner whenever feasible. This will also involve making changes into how the application is operated, optimized, and tested.

This report asserts that to design mission-critical applications in the cloud and achieve five nines end-to-end SLOs, enterprises must consider the following four domains:

1. *Multi-active architectures* as an architectural and deployment model
2. *Application reliability patterns* as a robust programming practice
3. *Site reliability engineering* as an improved operations culture
4. *Chaos engineering* as a testing and validation method

In Chapter 3, I cover multi-active architectures and application reliability patterns while providing actionable guiding principles. Additionally, I examine emerging software development patterns and other best practices that are relevant to each domain.

In Chapter 4, I explore how to govern and foster a culture of continuous improvement and innovation to operate Always On services. That chapter focuses on the principles of site reliability engineering (SRE) and the benefits of adopting chaos engineering as a testing practice to build confidence and provide evidence of a service's reliability.

CHAPTER 3
Always On Guiding Principles

Everything breaks.

This statement is a reminder that all systems fail and that it is essential to plan for all types of failures to minimize their impact and ensure continuity of service. The concept that everything will inevitably fail and the importance of planning for that failure is a common principle in many fields, not only IT. The idea is that by anticipating and planning for potential problems, we can design and build more reliable systems.

This principle is also known as *designing for failure* (DFF) which involves designing and building Always On services that can detect, tolerate, and sustain failures efficiently while transparently isolating the affected components without any human intervention. DFF serves as a guiding principle for decision making and execution, helping to ensure uniformity in methods and practices. This allows architects designing for Always On to apply this principle across every layer of the technology stack and operational model. It also enables a seamless and integrated approach to design, development, and deployment throughout all the components and elements that make up a mission-critical business service.

In the following section, I provide an overview of the cloud deployment methods that are compatible with an Always On approach with architectural principles that must be considered when designing and deploying mission-critical applications in the cloud, ensuring that they can make informed decisions that meet the specific needs of their workload.

Multi-Active Architectures

A crucial aspect of Always On is to shift the focus from recovering from a failure to transparently bypassing it. To achieve this, a multi-active approach is required. A multi-active architecture is the concept of deploying active workload instances consistently and redundantly across multiple locations. This distributes the capability of ensuring availability across multiple location scopes and balancing incoming traffic to process requests in parallel.

This in turn allows bypassing failures by proactively detecting failures, redirecting traffic, and enabling services to continue functioning seamlessly while failed components take the necessary time to recover. It is important to explain the necessary distinct concepts and technical capabilities that underpin a multi-active architecture:

- Location scopes
- Traffic management and service parallelism
- Transactional swimlanes and traffic affinity
- Share nothing, stretch nothing
- Deployment archetype

Let's dig into each of these in more detail.

Location Scopes

From a deployment perspective, applications must be deployed in multiple active location scopes capable of handling incoming traffic simultaneously. While organizations can create their own location scopes, either logically or physically, most cloud providers offer two established building blocks to address this: cloud regions and cloud zones.

- *Cloud regions* are geographical areas where a cloud provider has multiple data centers (or zones) that are isolated from each other but connected by high-speed networks.
- *Cloud zones*, often called availability zones, are separate data centers designed to provide redundancy and fault tolerance within a single cloud region so that if one zone experiences an outage or other issues, other zones can continue to operate normally.

Traditionally, a region includes a minimum of three zones. This is to supply in-region redundancy by providing infrastructure capacity that can allow workload instances to continue to operate and support the workload in case of zonal failures. This also helps minimize the risk of "split-brain" scenarios by providing a quorum majority needed for leader election and read/write majority for data consistency.

Cloud services provisioned in a region, whether it is a Kubernetes environment, a Kafka platform, or a database as a service instance, will typically cluster nodes in the three zones as a way to improve availability within a single region. This approach often involves stretching clusters across the three zones while maintaining a single shared logical control plane.

While there are important benefits to this approach, such as simplified management and monitoring, It is important to note that entire cloud region failures do[1] often[2] occur,[3] carrying significant risks that should not be disregarded.

It is also important to exercise caution in regard to the concept of "logical" availability zones that some cloud providers and on-prem solutions provide, which involves grouping virtual resources together in a logical manner to simulate the idea of physical zones. However, because logical availability zones are software-defined constructs and not physically separate infrastructures with distinct resources and fault domains, I do not subscribe to this idea.

Traffic Management and Service Parallelism

Traffic management and service parallelism are techniques used to process incoming traffic and transactions and distribute them across instances hosted on multiple active location scopes so that they can be processed in parallel. Incoming traffic must be balanced across location scopes to ensure that no scope becomes overloaded, improving scalability, performance, and reliability. This is true for both cloud zones and cloud regions.

1 "Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region," Amazon Web Services, 2017.

2 Google Cloud Incident #19006, status dashboard, November 2019.

3 Azure Status History, root cause analyses (RCAs) of previous service issues, March 2023.

DNS-based traffic steering and Anycast routing technologies direct incoming traffic across multiple cloud regions while zonal load balancers distribute traffic across the different availability zones, ensuring that failure bypass is enabled on a regional level and zonal level. It's important to pair multiple authoritative DNS and traffic management providers simultaneously, such as Cloudflare, Akamai, or NS1.

By combining both approaches and technologies shown in Figure 3-1, traffic is intelligently routed to the closest and most available cloud region, followed by the most performant and available cloud zone. This enables failure bypass at both regional and zonal levels, effectively eliminating single points of failure at every ingress point of entry. This is accomplished by continuously monitoring network conditions, user experience, transactional performance, and response time against real-time end-to-end SLOs to decide the optimal location scope for each request and transaction.

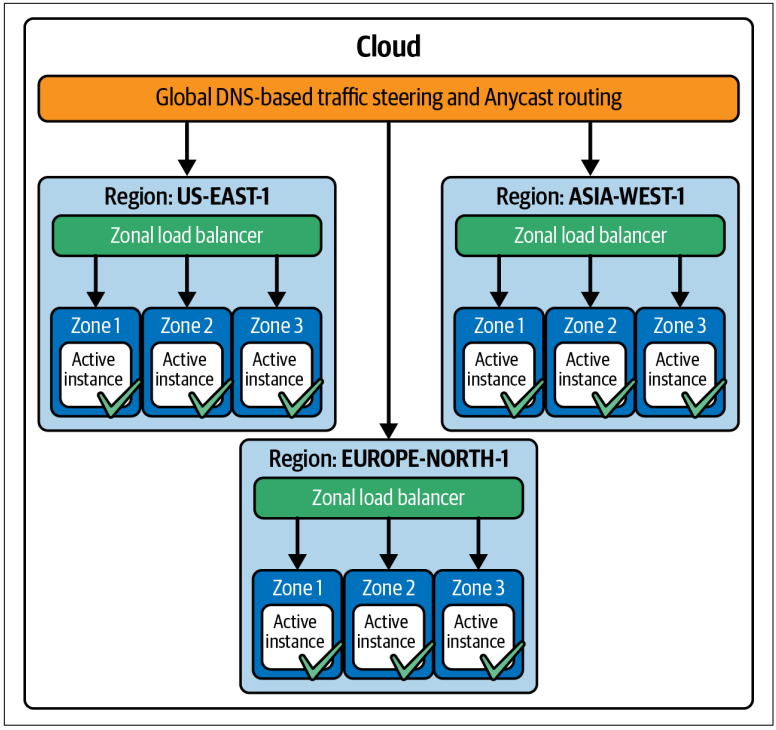

Figure 3-1. Global traffic steering and zonal load balancing

These capabilities will enable self-healing mechanisms to detect and bypass failures quickly with no human intervention. Any disruption

or failure in one location scope can be transparently bypassed to other locations that are still operational. Additionally, global traffic management can help organizations achieve better resource utilization, reduce network congestion, and provide better fault tolerance.

Transactional Swimlanes and Location Affinity

Because business services are active and served from multiple autonomous active cloud regions, it is important that data is stored and requests are processed separately across these regions. Each transaction flowing in a distinct region becomes its own *swimlane* and must be able to complete its specific task or function independently of other cloud regions. This means that traffic must not trombone (or zigzag) between regions.

Cloud architects and developers must then minimize the traffic that is sent back and forth between services in different regions, except for data replication and out-of-band management and monitoring tools. This approach involves placing related services and components in close proximity to each other within a region.

Client requests must have location affinity, meaning they must stick to the same location scope (region and zone) during the duration of related sessions. If the request fails in the middle of a transaction, it must gracefully restart in another location scope. This allows for a consistent and smooth user experience and transactional awareness, and it facilitates the accurate tracking of transactions based on their respective locations.

By logging transaction location scopes and incorporating them into the troubleshooting process, in the event of an error, the affected location scope is documented in distributed tracing and observability systems, thereby streamlining the troubleshooting process and promoting efficient problem resolution.

Share-Nothing, Stretch-Nothing

The *share-nothing, stretch-nothing* principle emphasizes the importance of distributing application instances and data across multiple isolated and independent locations, rather than extending or stretching them beyond a certain boundary.

This is important because stretching a system can increase the risk of failure propagation (and even security breaches) by extending

fault domains, which is often the case when using traditional tightly coupled infrastructure-based clusters or other tightly coupled systems that rely on shared resources, such as stretched storage volumes.

Expanding failure domains across cloud zones means that a single failure can cascade and impact multiple zones, potentially resulting in data loss and increased costs for recovery efforts. Additionally, stretched clusters can increase the risk of human errors or poorly executed configuration changes (for example, mistyping a faulty CLI command or pushing YAML files containing typos or errors) since any change made will impact all zones, leading to a regional outage.

There is no doubt that deploying clusters in general remains a fundamental aspect of cloud deployments. However, if a cluster is created and stretched across multiple zones, it is important to acknowledge that this cluster becomes a single failure domain. To maximize reliability, I recommend creating multiple active autonomous clusters across different zones, as shown in Figure 3-2.

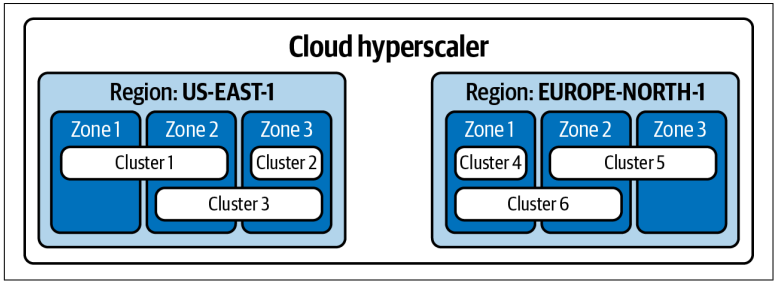

Figure 3-2. Multicluster deployments

In addition, Always On architectures mandate out-of-region deployments to ensure continuous availability and safeguard organizations from regional failures, providing enhanced reliability and ensuring that mission-critical services remain available to end users. This is also particularly beneficial for industries that must comply with business continuity and data governance regulations requiring out-of-region locations.

Deployment Archetype

As stated previously, the Always On deployment pattern should effectively restrict fault domains, provide service parallelism, and safeguard workloads against both zonal and regional failures. Additionally, the pattern must ensure zero downtime capabilities during both planned and unplanned outages, maintaining uninterrupted service and optimal performance.

Table 3-1 showcases some of the patterns that are commonly used in hyperscaler deployments. Patterns dependent on failover mechanisms are not deemed Always On compatible since they require complex and time-consuming recovery procedures that disrupt users and do not provide true zero downtime capabilities. Therefore, in this report, we will focus on the Always On globally distributed pattern to address these shortcomings.

Table 3-1. Deployment archetypes

Reliability patterns	Zonal failures protection	Regional failures protection	Zero downtime capable
Single region with multiple zones	Yes	No	No
Dual regions with failover region	Yes	Yes	No
Always On globally distributed	Yes	Yes	Yes

The Always On globally distributed deployment pattern requires a minimum of three active serving regions, known as the *3-active model*. This configuration ensures continuous service availability by distributing incoming traffic across all the regions, even during various disruptions, as it allows a location to be de-advertised and taken offline during unplanned and planned outages while still handling the anticipated maximum load with the remaining two locations.

As shown in Figure 3-3, each region must have at least two zones to prevent local zone failures from completely disrupting a region.

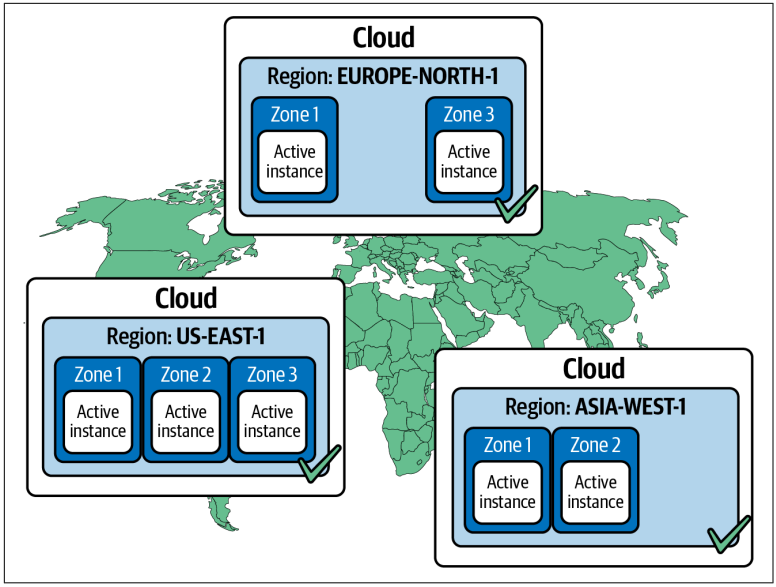

Figure 3-3. Always On globally distributed deployments

Although each region operates autonomously, their architecture and configurations are replicated across all three sites. GitOps and continuous deployment pipelines ensure consistent deployment and facilitate automation for rebuilds and infrastructure changes.

With global traffic management constantly monitoring and tracing the health, performance, and availability of all serving locations, it can effectively determine the unavailability of an application during an unplanned outage. If a location becomes unreachable, it is removed from the serving pool. Traffic is promptly steered to other serving locations, effectively bypassing the outage.

Operationally, the process is similar for planned outages. Operation teams de-advertise a serving location from the global traffic management pool before an application release or cloud maintenance task without affecting service-level objectives (SLOs) as traffic is seamlessly redirected to another active serving location until maintenance is complete.

All affected applications are verified before the serving location is re-advertised into the global traffic management pool. If any issues or concerns arise, the location remains out of service for problem

determination, resolution, and restoration to a previous functional state.

As an added advantage, adopting a 3-active model, shown in Figure 3-3, often proves to be more cost-effective and efficient than a 2-active approach:

- When utilizing two locations, each location must be provisioned at 100% capacity for cloud resources and software licenses to allow one location to handle the entire load if the other becomes inoperative, resulting in a 200% investment (100% + 100% = 200%).
- In contrast, using a 3-active model allows for a reduction in each location's capacity to 50%, resulting in an overall capacity of 150% rather than the 200% found in a 2-active setup, given that the required capacity is only 100% (50% + 50 % + 50% = 150%).

This reduction in capacity requirement not only lowers the overall investment but also enhances efficiency by allowing for greater flexibility in handling disruptions and distributing workload across the three locations.

As we move forward, I explore the significance of application development patterns in enhancing reliability, working in tandem with infrastructure and platforms to establish robust systems.

Application Reliability Patterns

I emphasized the significance of distributed deployment models in reinforcing the Always On architecture. By strategically deploying multi-active application instances and their dependencies, we can maintain availability in the face of zonal, regional, and both planned and unplanned outages. However, our reliability strategy doesn't stop there.

It is important to take a step up the stack and consider how the application itself can contribute to achieving even greater reliability with modern development patterns. Applications must be designed and architected to offer reliability by incorporating various design principles, architectural patterns, and best practices that address the common challenges of mission-critical services.

Cloud-native, microservices, and similar types of network-based applications manage numerous intertwined dependencies, encompassing a multitude of backend components, API and gRPC calls, data stores, and message queues. These components can give rise to a range of faults, from transient glitches to persistent issues, such as packet loss, network congestion, timeouts, database deadlocks, rate limiting, load balancing misrouting issues, and erroneous responses, among others. These can cascade to other critical parts of the system and cause catastrophic outages.

From an application perspective, it is necessary to embrace the same design for failure approach discussed earlier and consider these glitches as part of the overall end-to-end business service, identifying potential vulnerabilities in the application flow, as shown in Figure 3-4. This will help developers implement fallback mechanisms within the application stack that can significantly improve its reliability and ensure a more robust and antifragile system.

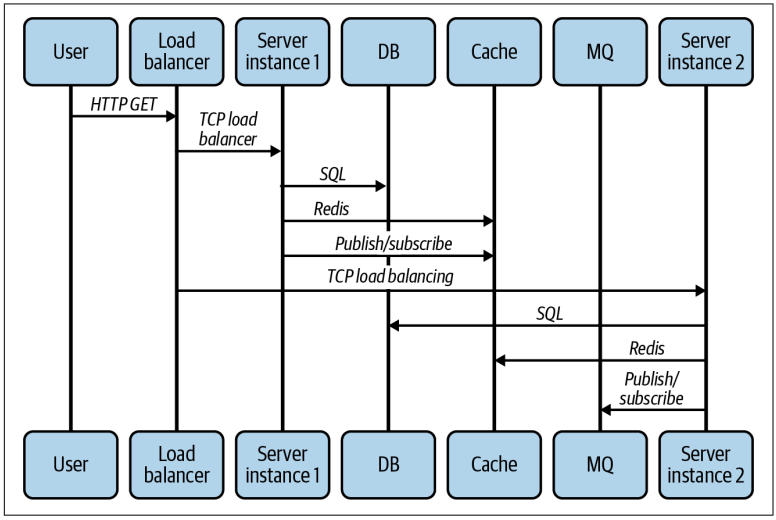

Figure 3-4. Simplified example of an application flow

Similar to the deployment models introduced in the earlier sections, it is essential for applications to proactively bypass faults, self-heal when necessary, ensure reliable message delivery and order, and safeguard data consistency and integrity.

To do that, Always On applications need to incorporate the following patterns that developers should utilize from modern

cloud-native development frameworks, including but not limited to Spring, MicroProfile, Quarkus, and specialized fault tolerance libraries such as Resilience4j, Hystrix, and Polly:

- Fault isolation
- Loose coupling
- Flow management and control
- Health check endpoints and supervision
- Location awareness
- Data and transactional consistency

Let's dig into each of these in more detail.

Fault Isolation

Fault isolation plays a crucial role in preventing cascading failures among microservices and other types of API-based and network-based services. By isolating components, failures in one part of the system do not spread to other areas. Various techniques can be employed to achieve fault isolation, such as:

Process boundaries
: This isolation technique involves separating components or microservices into distinct processes, each running within its own memory space (or runtime). This separation ensures that a failure in one process does not directly affect others, as they are independently managed by individual containers or operating systems.

Circuit breakers
: The circuit breaker pattern is designed to prevent cascading failures by monitoring the health of a dependent service and temporarily disabling communication with it if it becomes unresponsive or experiences excessive errors. Once the service recovers, the circuit breaker resets, and communication is reestablished.

Bulkheads
: Inspired by the design of ships, bulkheads are partitions that segregate components or services within an application by isolating distinct functional areas or resources and preventing overload and failures in one resource from propagating

to others. Examples of bulkheads are thread pool bulkheads, semaphore bulkheads, and database pool bulkheads.

Loose Coupling

Loose coupling involves creating applications that maintain essential functionality even when individual components or services encounter issues. The goal is to decouple services (making them loosely coupled) and minimize the impact of failures in the business and affected users, while ensuring that other users can continue to access the functioning services without disruption.

In a loosely coupled system, components interact through well-defined interfaces and protocols, diminishing the chances of one component's changes or failures affecting others. This pattern also simplifies individual component scaling and deployment without disrupting the entire application. A range of loose coupling techniques can then be utilized, such as:

Asynchronous communication
: This technique refers to allowing components to process and respond to requests at their own pace without time limitations. By relaxing temporal constraints, components can handle occasional slow responses without causing the entire application to fail. It allows components to continue processing other tasks without being blocked by delayed responses from other services.

Event-driven architecture
: An event-driven architecture involves components communicating through queued events (or actionable facts) instead of direct calls to one another. This approach promotes decoupling by allowing components to react to ordered events independently, reducing the impact of failures on the overall system.

Idempotency
: Idempotent operations can be executed multiple times without causing any side effects or unintended consequences. This means that services and components can safely retry operations in case of failure.

Flow Management and Control

Managing flow and overall requests latency can significantly impact application performance and user experience. Flow control involves implementing strategies to minimize the response time of an application by reducing delays and managing API and gRPC traffic flow and timeouts. Developers can leverage the following techniques to manage the flow of requests or messages in a system:

Bounded queues
: Bounded queues help manage the number of requests or messages waiting to be processed, preventing an uncontrolled growth of pending items that could lead to resource exhaustion or slow response times. By limiting the queue size, systems can maintain a balance between incoming requests and processing capacity. This pattern is especially useful when handling workloads with varying demand or when dealing with components that have different processing capabilities.

Load shedding and throttling
: Load shedding and throttling are two techniques used to manage and control traffic in a system. Load shedding involves deliberately reducing incoming traffic to maintain the stability and responsiveness of the entire system during periods of high demand or component failure. On the other hand, throttling involves limiting the processing rate of traffic by a system or component, based on properties of individual resources. Unlike throttling, load shedding proactively avoids overloading the system by limiting requests based on the overall state of the system rather than individual resource properties.

Timeouts, retries, and backoffs
: Timeouts are employed to limit the waiting time for a response from a dependent service or component. By setting a reasonable timeout, services can prevent long waits and ensure that slow or unresponsive components do not cause the entire application to stall. Similarly, an exponential backoff strategy is used in conjunction with retries to introduce a delay between retry attempts to alleviate the risk of overloading a system or service with repeated requests by giving it time to recover from transient issues.

Fail-fast and fallback
: The fail-fast and fallback approach involves detecting errors or failures early in the process and transparently falling back the operation immediately to an alternative path, rather than attempting to proceed in a degraded system state. This technique helps reduce latency by preventing wasted time and resources on processing that is likely to fail.

Health Check Endpoints and Supervision

Health check endpoints refers to the supervision, monitoring, and detection of the health and status of the desired end-to-end SLO and the overall behavior of the business service. This is essential to enable gatekeeping mechanisms and traffic management between application interfaces and endpoints and take appropriate actions as needed.

Health check endpoints typically respond with an HTTP status code and, optionally, a JSON or XML payload containing more detailed information about the service's health. The primary purpose of health check endpoints is to enable external supervising systems and orchestrators (such as global traffic managers, load balancer, circuit breakers, and synthetic testing and monitoring systems) to make informed decisions about routing traffic and scaling instances.

Common types of health checks in Always On architectures include:

Liveness and readiness checks
: These tests are designed to indicate whether a service or component is up and running. If a liveness probe fails, it signals that the instance has crashed or is experiencing a failure, and the orchestrator (such as a Kubernetes scheduler) may decide to restart the instance. A readiness probe, on the other hand, is used to determine if a service is fully ready to receive and process incoming requests. This type of health check can be useful during runtime and JVM startups, when a service may be running but not yet fully initialized and temporarily unable to handle requests.

Simulated transactional tests
: These synthetic tests simulate complex user interactions or multistep processes within an application, API, or website, such as completing a purchase, creating an account, or filling out a form. Transactional checks can be combined with simulation

originating from browser-based and mobile app-based testing execution environments. These tests measure the time it takes for an API or a process to respond to a request. Response time monitoring helps identify potential performance bottlenecks and ensure that the system meets specified performance requirements.

Location Awareness

Location awareness refers to the ability of a service to determine its geographical and logical location scope or even the network proximity of its components, resources, data, and users and make informed decisions based on this information. Examples of location-awareness attributes could be servers, server racks, data centers, cloud zones, cloud regions, and cluster or geographical origination (a method commonly used for data sharding). Location information can be used to optimize various aspects of the application, such as VM or worker node affinity/anti-affinity rules, read/write bias (CQRS pattern), replica placement, unique identifiers generation, leadership election and balancing, and even go as far as deciding on SQL primary keys or manipulating explicit `INSERT INTO` statements.

Location awareness also helps optimize the strategic placement of data replicas across different geographical locations to reduce the risk of data loss while allowing applications to process data closer to where it is generated or consumed.

An additional advantage of location awareness is that it enhances overall system security by directing traffic to suitable geographic locations, ensuring compliance with regional security and data residency policies, which is often crucial when maintaining mission-critical services.

Data and Transactional Consistency

Preserving data consistency and transactional integrity by itself can be a complex task. Making sure data stays consistent is key and can be pretty challenging to get right, especially when applications have to deal with accessing, writing, and processing data from several active locations simultaneously. This complexity necessitates careful consideration and thorough analysis by architects and developers to fulfill possible (and acceptable) consistency level and parallel data access requirements of an Always On application.

There are various data consistency patterns that go beyond the commonly known synchronous (strong consistency), asynchronous (eventual consistency), and semisynchronous replication patterns. These are employed to address the trade-offs highlighted by the CAP theorem,[4] which states that a distributed system cannot simultaneously offer consistency, availability, and partition tolerance:

Replication patterns
: Different data replication patterns can be applied to cater to specific needs of an application in terms of availability, consistency, and latency. Notable patterns include primary/secondary, multimaster, peer-to-peer, sharding, and quorum consensus with leader election. Numerous NewSQL solutions (AWS DynamoDB, Google Spanner, Microsoft Azure Cosmos DB, CockroachDB, MongoDB, Apache Cassandra, etc.) and modernized traditional DBMS implementations (Percona Server for MySQL, Apache ShardingSphere, Oracle Grid, etc.) incorporate these patterns. Other out-of-band change data capture (CDC) technologies such as IBM Q Replication, Oracle GoldenGate, Debezium and even Kafka help traditional database systems offer similar capabilities.

Tunable consistency
: Tunable data consistency allows architects to make informed decisions about the trade-offs between consistency and latency by providing the ability to tune different consistency levels[5] that can be chosen based on the nature of the operation (read or write), application, and even users. For example, databases like Cosmos DB and Cassandra offer multiple consistency levels that can be configured on a per-operation basis such as strong, session-based, consistent-prefix, etc. Additionally, streaming platforms like Kafka provide multiple delivery guarantees such as *at least once* and *exactly once*.

Conflict resolution
: In multimaster (read/write anywhere) or peer-to-peer replication scenarios, conflicts may arise when multiple nodes update the same data concurrently. Conflict resolution strategies like

4 "CAP Theorem," Wikipedia, last modified April 27, 2023.

5 Robson A. Campêlo et al., "A Brief Survey on Replica Consistency in Cloud Environments," *Springer Nature*, February 21, 2020.

last-write-wins, log and notify, merge, version vectors, CRDTs, or other application-specific logic can be employed to resolve these conflicts and maintain data integrity and consistency. Conflict resolution and reconciliation are inevitable parts of distributed systems, and they should not be intimidating. Instead, they should be viewed as necessary mechanisms to handle inconsistencies that arise in the system.

Distributed transaction protocols
: Transactions ensure that all operations within a transaction are either fully completed or fully rolled back, maintaining the integrity of the data. To achieve this, developers can leverage two-phase commits (2PC), which ensures strong messaging consistency across multiple resources (e.g., databases, message queues) and Saga transactions that let applications perform a sequence of local transactions, each executed by a different resource and executing compensating transactions in case one fails to undo the effects of the preceding successful operation, effectively rolling back the entire Saga.

Caching
: Caching assists applications to maintain responsiveness and availability even during periods of instability and transient issues by displaying stale but relevant data while simultaneously accepting modifications that will later be synchronized with primary data stores. Read-through, write-through, and write-back caching strategies must be evaluated, while tools like AWS DynamoDB Accelerator (DAX) and Red Hat Data Grid enable developers to effortlessly integrate these caching mechanisms into their applications.

Global clocks and order
: The order of messages is crucial for effective messaging and queuing systems, requiring accurate and consistent timestamping regardless of sender and receiver locations. Services such as AWS Time Sync Service and GCP TrueTime API utilize technologies such as redundant satellite-connected servers and GPS-aware atomic clocks to provide a stable UTC reference for high accuracy and reliability in timekeeping.

Naturally, latency remains the proverbial elephant in the room. It's the silent killer that can bring even the most well-designed system to its knees. This is why taking a thoughtful business perspective

approach to data consistency design early on is important to optimize the specific needs of the application.

The foundational architectures and guiding principles that we discussed above are essential for laying the groundwork to achieve an Always On state, but the technology aspect is only one piece of the puzzle. In the next chapter, we will explore how an Always On program's success also depends on a distinctive governance model, organizational restructuring, and a transformative cultural shift.

CHAPTER 4

Culture, Governance, and Organization

As previously discussed, striving to be Always On demands strategic decision making from the highest levels of an organization. This involves adopting a modern and unique approach of governing and a new method of operations.

Mature CIO organizations establish business service classification review boards to determine the appropriate service tier and the end-to-end SLO required for a specific mission-critical business service. They consider factors such as service criticality, continuous reliability expectations, development time, operational expenses, and the financial consequences of potential outages.

When organizations discuss promoting the need for reliable services, they often encounter doubts or inquiries about the feasibility of achieving it from both business and technical perspectives. It is vital, then, to foster a culture that embraces continuous ops, or as it is known by its newer name, site reliability engineering (SRE), a framework that brings together business, architects, application owners, development, and operations teams.

Site reliability engineers evaluate and approve proposals for implementing Always On for a nominated business service after the need for Always On has been socialized organization-wide. They conduct a fit analysis to ensure alignment with existing Always On patterns and guiding principles, accelerating the delivery of customer-focused reliable services and improving decision making.

In this chapter, I briefly describe SRE and chaos engineering, concepts that are instrumental not only in operating mission-critical services but also in testing, validating, and building confidence in their reliability.

Site Reliability Engineering

Site reliability engineering, pioneered by Google, is an engineering discipline that focuses on applying software engineering principles and practices to operations. In an Always On environment, SREs are tasked with establishing and adhering to end-to-end SLOs, which often entails striking a balance between development velocity (such as new deployments, changes, and new application features) and reliability, all while maintaining a customer-centric mindset. SREs implement automation, observability, testing, and incident response strategies to ensure that applications meet these objectives and continuously improve over time.

The journey toward adopting SRE starts with a cultural shift in the organization, emphasizing the importance of a blameless postmortem culture. This approach focuses on learning from incidents and improving systems rather than assigning blame. By fostering collaboration among development, operations, and other crossfunctional teams, it becomes easier to dismantle silos and align everyone's efforts toward common objectives. Ultimately, an SRE culture cultivates an approach of continuous improvement, enabling teams to proactively detect and address potential reliability concerns at an early stage.

When managing Always On services, SRE teams must have a comprehensive understanding of both the functioning and potential failure points of the end-to-end application flow and underlying infrastructure. This means that the SRE team responsible for lifecycle operations should be actively involved in the design, implementation, testing, and validation stages. This involvement ensures prompt feedback and corrective measures if the design fails to meet all the guiding principles for Always On.

SRE is a wide topic. I recommend exploring the literature that Google regularly maintains for a deeper understanding. However, for the scope of this report, I will focus on the following specific areas:

- Build to manage
- Error budgets
- Observability and proactive service management
- Declarative and continuous deployment
- Graceful location scope–based updates

Let's explore each of these in more detail.

Build to Manage

Build to manage refers to the practice of designing and developing systems with manageability, observability, and reliability as integral components from the very beginning. It encompasses a collection of manageability features incorporated within the application as part of the development and release process.

The build-to-manage approach deserves an extensive and detailed examination. While I will highlight some of its crucial aspects, it is necessary to understand its related concepts and strategies:[1]

Log format and catalog
: Adhering to the practice of composing well-structured log messages is crucial in order to capture pertinent and consistent information during runtime. Logs should effectively convey the who, when, where, and what, along with a severity ranking and a well-defined timestamp.

Deployment correlation
: By utilizing deployment markers, it is possible to indicate deployment activities on the same chart or timeline that displays reliability metrics. This approach enables SREs to visually correlate reliability issues with recent deployment of a new version, making it easier to identify the cause of any issues.

Runbooks and knowledge base
: A knowledge base serves as a central repository for storing any information and runbooks relevant to troubleshooting or resolving issues related to an incident. Entries may include troubleshooting runbooks and steps taken by SREs during a

1 Ingo Averdunk, "Build to Manage," IBM Cloud Garage and Solution Engineering (GSE), December 23, 2019.

resolution process. However, runbooks must be automated and executed by first responders, rather than simply documentation.

Concurrent versioning
: Multi-instances deployments enable running distinct versions of an application in separate location scopes, facilitating canary testing and collaborative database schema changes. For this to safely work, the use of feature flags can be leveraged, a technique that allows SREs and developers to turn specific features of an application on or off without changing the codebase.

Error Budgets

SRE incorporates the concept of *error budgets* to balance the goals of frequent, agile, and rapid development and maintain reliability. Error budgets provide a quantitative measure to manage risk and make informed decisions about stability, feature development, and deployment velocity.

An error budget is derived from the end-to-end SLO. For example, if a service has an end-to-end SLO of 99.99%, it means that it has a 0.01% allowable downtime, which is the error budget. This is then used to drive decisions and trade-offs between different teams in an organization. For example, if a service is well within its error budget, the development team may decide to push new features more aggressively. Alternatively, if the service is approaching or exceeding its error budget, the development team might slow down new releases and focus on improving stability and reliability instead.

By utilizing error budgets, SREs take responsibility (and accountability) for upholding the various SLOs in place.

Observability and Proactive Service Management

Observability is a key aspect of managing Always On applications, as it provides the necessary visibility into application behavior and performance. Observability practices include collecting and analyzing logs, metrics, and transactional traces from applications, infrastructure components, and transactions to identify trends, detect anomalies, and uncover the root causes of issues.

Traditionally, operations may only monitor infrastructure services and respond to failures, but SREs are paranoid. They monitor all aspects of a business service's reliability and performance, both

internally and externally. By monitoring trending and correlating every aspect of the business service, abnormalities can be detected before they lead to incidents and problems. This approach represents a significant shift from reactive to proactive service management. If a user or business client reports an issue that the team is unaware of, SREs consider their mission a failure.

Declarative and Continuous Deployment

GitOps is a modern approach to managing infrastructure and application deployments using Git as the single source of truth. By defining infrastructure and deployment configurations declaratively as code using tools like Crossplane, Terraform, and Ansible, stored in a version-controlled repository, GitOps enables automated, self-documented, and auditable continuous workload deployment.

GitOps empowers SREs to streamline deployments by writing code once and reusing it for multiple deployments. Deploying a new application or updating an existing one simply involves updating the repository. This approach helps ensure consistency across multi-active environments and reduces the risk of human error while performing changes across location scopes.

Graceful Location Scope–Based Updates

Location scope–based updates, or "one region at a time," involve the gradual execution of planned changes or application releases by SREs to ensure zero downtime, similar to blue-green and canary deployment methods.

Before releasing an application or performing a maintenance job, SREs de-advertise a location scope (such as a cloud region) from the global traffic management pool to gracefully shutdown without causing errors to clients. Sometimes, SREs must also know the order of the components to shut down first, by reverse shutting down the dependency injection of a component, which allows them to work on their tasks without impacting SLOs and losing in-transit processes and data.

Once maintenance is completed or a new version of the application has been successfully pushed, all affected applications are verified and tested before the serving location is reintegrated into the global traffic management pool. These steps are replicated to other regions, one at a time, effectively allowing organizations to consistently

deploy new application features and perform maintenance tasks during working hours and peak times.

To ensure the availability of a business service, SREs need confidence in its reliability. In the next section I discuss chaos engineering, a practice that is essential to test and validate end-to-end SLOs.

Chaos Engineering

Chaos engineering is a proactive and principled practice of conducting reliability experiments on a business service to strengthen its ability to withstand unpredictable and unforeseen conditions and to uncover issues that might not be detected during preproduction testing. This is achieved by intentionally injecting faults and errors into a component to observe how it behaves under adverse conditions. Examples of failure injections include node/pod failure, packet drops, cloud zone and region failures, latency, I/O delays, process kill, disk fill, certificate expiry, and clock changes, among others.

Contrary to what the name might imply, these experiments are carefully designed and orchestrated and follow a rigid method, as shown in Figure 4-1. The process begins with understanding the system end to end, identifying potential weaknesses, followed by formulating a hypothesis based on these observations.

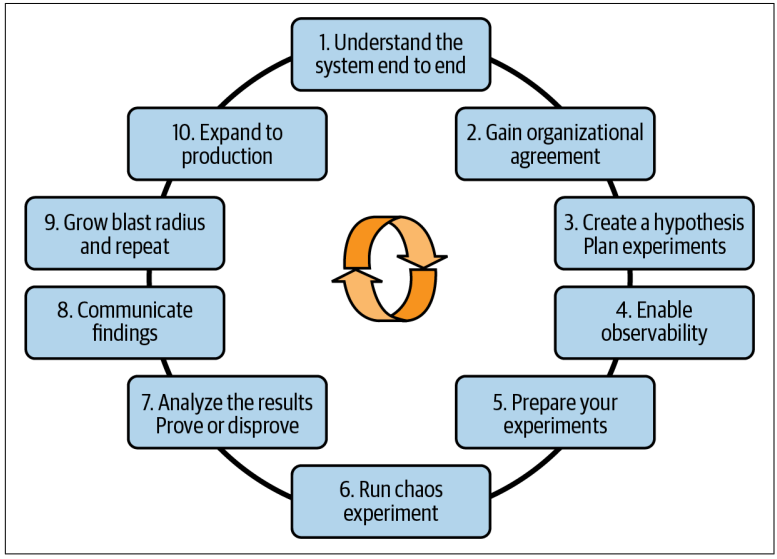

Figure 4-1. Chaos engineering methodology

Subsequently, a tailored experiment is planned and executed against the system. As previously discussed, SREs must have a comprehensive understanding of the entire technology stack and of the end-to-end application flow. This will help them to effectively experiment on all components.

It is important however, to begin simulating realistic scenarios by injecting likely failures and bugs and then steadily escalating complexity. For example, if latency has been an issue in the past, SREs can intentionally introduce faults that cause latency to occur. The results of the experiment are then measured and compared with the hypothesis to either confirm or refute it. By doing so, they can identify immediate weaknesses in the system and develop strategies to mitigate or eliminate them before focusing on more complex failure scenarios.

It's essential that impact is contained to avoid cascading failures and to minimize blast radiuses. This can be achieved by targeting only a select group of services, possibly in a cloud region that's been de-advertised by the SREs, or by ensuring that dependent services are gracefully disconnected. The use of feature flags and canary releases can also be helpful to limit the impact of the experiment to only a small group of users. It is also important to have workable rollback plans in place in case an experiment goes wrong if deemed necessary.

These experiments are typically conducted during designated "game days" to assess a service's behavior in a controlled setting. As the organization matures and becomes more experienced, experiments should be conducted gradually in production to ensure the application's robustness when it matters. Experiments can then be automated and continuously executed or as part of continuous delivery pipelines.

SREs can use chaos engineering tools available by cloud providers, including AWS Fault Injection Simulator and Azure Chaos Studio to experiment on cloud-native managed services. In addition, cloud-agnostic tools like Gremlin, Reliably, Chaos Toolkit, and LitmusChaos can perform more intricate experiments that involve multiple clouds and platforms.

Observability tools must be well configured to learn from relevant results and insights, enabling architectural improvement, issuing bug reports, or submitting feature requests. To that end, SREs must

maintain open communication throughout the process, fostering a culture that appreciates the benefits of introducing controlled, short-term risks to enhance overall long-term reliability.

By simulating experiments and identifying issues proactively, organizations can identify and remediate architectural weaknesses, thus improving service reliability and stability early on. This will in turn provide confidence and evidence that their mission-critical services are meeting their end-to-end SLOs.

Closing Words

Achieving Always On is an attainable objective that often involves a lengthy journey. An entire organization must fully embrace this journey and accept the necessary cultural changes to their business. A holistic approach is required, encompassing people, processes, and technology, and it cannot be achieved by changing only one of the three.

As an iterative process, the Always On transformation is marked by a continuous cycle of planning, execution, evaluation, and improvement. To achieve this, organizations need to comprehensively address all aspects of a client's journey and assess reliability and weaknesses across the end-to-end architecture. This includes embracing a designing-for-failure mindset not only for the infrastructure, platforms, and services but also for the applications themselves.

Transforming operations and testing methodologies is also a crucial aspect of managing mission-critical services by adopting site reliability engineering and chaos engineering principles. By focusing on these areas and continuously iterating and improving based on feedback, organizations can proactively identify and address any potential issues, ultimately enhancing the overall reliability of their business services.

By adhering to the guiding principles described in this report, organizations can lay a robust foundation for designing, developing, deploying, and managing mission-critical workloads in the cloud and succeed at adopting an Always On strategy, ensuring a seamless and reliable experience for their clients throughout their journey.

About the Author

Haytham Elkhoja is a principal architect at Kyndryl, office of the CTO. He led the incubation of Always On as a global consultancy and established the Chaos Engineering Guild at IBM and then at Kyndryl. Haytham leads cloud reliability transformation engagements around the world, focusing on industries such as banking and finance, energy, defense, and transportation.

Haytham is an avid golfer and snowboarder.