Reliability Pillar

AWS Well-Architected Framework

Reliability Pillar: AWS Well-Architected Framework

Table of Contents

Abstract and Introduction 1
- Abstract 1
- Introduction 1

Reliability 2
- Design Principles 2
- Definitions 2
  - Resiliency, and the Components of Reliability 3
  - Availability 3
  - Disaster Recovery (DR) Objectives 6
- Understanding Availability Needs 7

Foundations 8
- Manage Service Quotas and Constraints 8
  - Resources 9
- Plan your Network Topology 9
  - Resources 13

Workload Architecture 14
- Design Your Workload Service Architecture 14
  - Resources 16
- Design Interactions in a Distributed System to Prevent Failures 16
  - Resources 18
- Design Interactions in a Distributed System to Mitigate or Withstand Failures 19
  - Resources 23

Change Management 24
- Monitor Workload Resources 24
  - Resources 27
- Design your Workload to Adapt to Changes in Demand 27
  - Resources 29
- Implement Change 29
  - Additional deployment patterns to minimize risk: 31
  - Operational Readiness Reviews (ORRs) 31
  - Resources 32

Failure Management 33
- Back up Data 33
  - Resources 34
- Use Fault Isolation to Protect Your Workload 35
  - Resources 39
- Design your Workload to Withstand Component Failures 40
  - Resources 42
- Test Reliability 43
  - Resources 46
- Plan for Disaster Recovery (DR) 46
  - Resources 48

Example Implementations for Availability Goals 50
- Dependency Selection 50
- Single-Region Scenarios 50
  - 2 9s (99%) Scenario 51
  - 3 9s (99.9%) Scenario 52
  - 4 9s (99.99%) Scenario 54
- Multi-Region Scenarios 56
  - 3½ 9s (99.95%) with a Recovery Time between 5 and 30 Minutes 56
  - 5 9s (99.999%) or Higher Scenario with a Recovery Time under 1 minute 59
- Resources 61
  - Documentation 61
  - Labs 62

External Links 62
Books 62
Conclusion 63
Contributors 64
Further Reading 65
Document Revisions 66
Appendix A: Designed-For Availability for Select AWS Services 68

Reliability Pillar - AWS Well-Architected Framework

Publication date: **July 2020** (*Document Revisions* (p. 66))

Abstract

The focus of this paper is the reliability pillar of the AWS Well-Architected Framework. It provides guidance to help customers apply best practices in the design, delivery, and maintenance of Amazon Web Services (AWS) environments.

Introduction

The AWS Well-Architected Framework helps you understand the pros and cons of decisions you make while building workloads on AWS. By using the Framework you will learn architectural best practices for designing and operating reliable, secure, efficient, and cost-effective workloads in the cloud. It provides a way to consistently measure your architectures against best practices and identify areas for improvement. We believe that having well-architected workload greatly increases the likelihood of business success.

The AWS Well-Architected Framework is based on six pillars:

- Operational Excellence
- Security
- Reliability
- Performance Efficiency
- Cost Optimization
- Sustainability

This paper focuses on the reliability pillar and how to apply it to your solutions. Achieving reliability can be challenging in traditional on-premises environments due to single points of failure, lack of automation, and lack of elasticity. By adopting the practices in this paper you will build architectures that have strong foundations, resilient architecture, consistent change management, and proven failure recovery processes.

This paper is intended for those in technology roles, such as chief technology officers (CTOs), architects, developers, and operations team members. After reading this paper, you will understand AWS best practices and strategies to use when designing cloud architectures for reliability. This paper includes high-level implementation details and architectural patterns, as well as references to additional resources.

Reliability

The reliability pillar encompasses the ability of a workload to perform its intended function correctly and consistently when it's expected to. This includes the ability to operate and test the workload through its total lifecycle. This paper provides in-depth, best practice guidance for implementing reliable workloads on AWS.

Topics

- Design Principles (p. 2)
- Definitions (p. 2)
- Understanding Availability Needs (p. 7)

Design Principles

In the cloud, there are a number of principles that can help you increase reliability. Keep these in mind as we discuss best practices:

- **Automatically recover from failure:** By monitoring a workload for key performance indicators (KPIs), you can trigger automation when a threshold is breached. These KPIs should be a measure of business value, not of the technical aspects of the operation of the service. This allows for automatic notification and tracking of failures, and for automated recovery processes that work around or repair the failure. With more sophisticated automation, it's possible to anticipate and remediate failures before they occur.
- **Test recovery procedures:** In an on-premises environment, testing is often conducted to prove that the workload works in a particular scenario. Testing is not typically used to validate recovery strategies. In the cloud, you can test how your workload fails, and you can validate your recovery procedures. You can use automation to simulate different failures or to recreate scenarios that led to failures before. This approach exposes failure pathways that you can test and fix *before* a real failure scenario occurs, thus reducing risk.
- **Scale horizontally to increase aggregate workload availability:** Replace one large resource with multiple small resources to reduce the impact of a single failure on the overall workload. Distribute requests across multiple, smaller resources to ensure that they don't share a common point of failure.
- **Stop guessing capacity:** A common cause of failure in on-premises workloads is resource saturation, when the demands placed on a workload exceed the capacity of that workload (this is often the objective of denial of service attacks). In the cloud, you can monitor demand and workload utilization, and automate the addition or removal of resources to maintain the optimal level to satisfy demand without over- or under-provisioning. There are still limits, but some quotas can be controlled and others can be managed (see Manage Service Quotas and Constraints (p. 8)).
- **Manage change in automation**: Changes to your infrastructure should be made using automation. The changes that need to be managed include changes to the automation, which then can be tracked and reviewed.

Definitions

This whitepaper covers reliability in the cloud, describing best practice for these four areas:

- Foundations
- Workload Architecture

- Change Management
- Failure Management

To achieve reliability you must start with the foundations—an environment where service quotas and network topology accommodate the workload. The workload architecture of the distributed system must be designed to prevent and mitigate failures. The workload must handle changes in demand or requirements, and it must be designed to detect failure and automatically heal itself.

Topics

- Resiliency, and the components of Reliability (p. 3)
- Availability (p. 3)
- Disaster Recovery (DR) Objectives (p. 6)

Resiliency, and the components of Reliability

Reliability of a workload in the cloud depends on several factors, the primary of which is *Resiliency*:

- **Resiliency** is the ability of a workload to recover from infrastructure or service disruptions, dynamically acquire computing resources to meet demand, and mitigate disruptions, such as misconfigurations or transient network issues.

The other factors impacting workload reliability are:

- Operational Excellence, which includes automation of changes, use of playbooks to respond to failures, and Operational Readiness Reviews (ORRs) to confirm that applications are ready for production operations.
- Security, which includes preventing harm to data or infrastructure from malicious actors, which would impact availability. For example, encrypt backups to ensure that data is secure.
- Performance Efficiency, which includes designing for maximum request rates and minimizing latencies for your workload.
- Cost Optimization, which includes trade-offs such as whether to spend more on EC2 instances to achieve static stability, or to rely on automatic scaling when more capacity is needed.

Resiliency is the primary focus of this whitepaper.

The other four aspects are also important and they are covered by their respective pillars of the AWS Well-Architected Framework. Many of the best practices here also address those aspects of reliability, but the focus is on resiliency.

Availability

Availability (also known as *service availability*) is both a commonly used metric to quantitatively measure resiliency, as well as a target resiliency objective.

- **Availability** is the percentage of time that a workload is available for use.

Available for use means that it performs its agreed function successfully when required.

This percentage is calculated over a period of time, such as a month, year, or trailing three years. Applying the strictest possible interpretation, availability is reduced anytime that the application isn't operating normally, including both scheduled and unscheduled interruptions. We define *availability* as follows:

$$Availability = \frac{Available\ for\ Use\ Time}{Total\ Time}$$

- Availability is a percentage uptime (such as 99.9%) over a period of time (commonly a month or year)
- Common short-hand refers only to the "number of nines"; for example, "five nines" translates to being 99.999% available
- Some customers choose to exclude scheduled service downtime (for example, planned maintenance) from the *Total Time* in the formula. However, this is not advised, as your users will likely want to use your service during these times.

Here is a table of common application availability design goals and the maximum length of time that interruptions can occur within a year while still meeting the goal. The table contains examples of the types of applications we commonly see at each availability tier. Throughout this document, we refer to these values.

Availability	Maximum Unavailability (per year)	Application Categories
99% (p. 51)	3 days 15 hours	Batch processing, data extraction, transfer, and load jobs
99.9% (p. 52)	8 hours 45 minutes	Internal tools like knowledge management, project tracking
99.95% (p. 56)	4 hours 22 minutes	Online commerce, point of sale
99.99% (p. 54)	52 minutes	Video delivery, broadcast **workloads**
99.999% (p. 59)	5 minutes	ATM transactions, telecommunications **workloads**

Measuring availability based on requests. For your service it may be easier to count successful and failed requests instead of "time available for use". In this case the following calculation can be used:

$$Availability = \frac{Successful\ Responses}{Valid\ Requests}$$

This is often measured for one-minute or five-minute periods. Then a monthly uptime percentage (time-base availability measurement) can be calculated from the average of these periods. If no requests are received in a given period it is counted at 100% available for that time.

Calculating availability with hard dependencies. Many systems have hard dependencies on other systems, where an interruption in a dependent system directly translates to an interruption of the

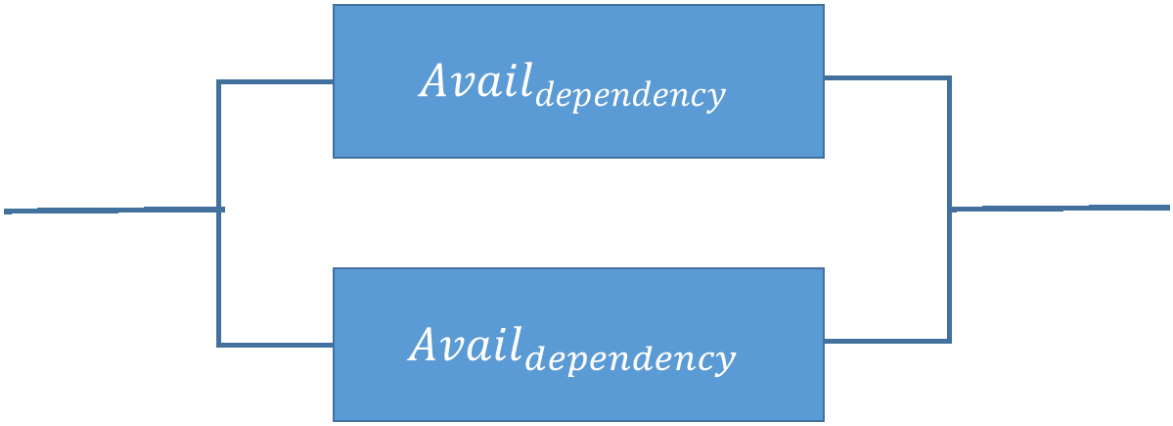
invoking system. This is opposed to a soft dependency, where a failure of the dependent system is compensated for in the application. Where such hard dependencies occur, the invoking system's availability is the product of the dependent systems' availabilities. For example, if you have a system designed for 99.99% availability that has a hard dependency on two other independent systems that each are designed for 99.99% availability, the workload can theoretically achieve 99.97% availability:

$Avail_{invok} \times Avail_{dep1} \times Avail_{dep2} = Avail_{workload}$

99.99% × 99.99% × 99.99% = 99.97%

It's therefore important to understand your dependencies and their availability design goals as you calculate your own.

Calculating availability with redundant components. When a system involves the use of independent, redundant components (for example, redundant resources in different Availability Zones), the theoretical availability is computed as 100% minus the product of the component failure rates. For example, if a system makes use of two independent components, each with an availability of 99.9%, the effective availability of this dependency is 99.9999%:



$Avail_{effective} = Avail_{MAX} - ((100\% - Avail_{dependency}) \times (100\% - Avail_{dependency}))$

99.9999% = 100% − (0.1%×0.1%)

Shortcut calculation: If the availabilities of all components in your calculation consist solely of the digit nine, then you can sum the count of the number of nines digits to get your answer. In the above example two redundant, independent components with three nines availability results in six nines.

Calculating dependency availability. Some dependencies provide guidance on their availability, including availability design goals for many AWS services (see Appendix A: Designed-For Availability for Select AWS Services (p. 68)). But in cases where this isn't available (for example, a component where the manufacturer does not publish availability information), one way to estimate is to determine the **Mean Time Between Failure (MTBF)** and **Mean Time to Recover (MTTR)**. An availability estimate can be established by:

$$Avail_{EST} = \frac{MTBF}{MTBF + MTTR}$$

For example, if the MTBF is 150 days and the MTTR is 1 hour, the availability estimate is 99.97%.

For additional details, see Availability and Beyond: Understanding and improving the resilience of distributed systems on AWS, which can help you calculate your availability.

Costs for availability. Designing applications for higher levels of availability typically results in increased cost, so it's appropriate to identify the true availability needs before embarking on your application design. High levels of availability impose stricter requirements for testing and validation under exhaustive failure scenarios. They require automation for recovery from all manner of failures, and require that all aspects of system operations be similarly built and tested to the same standards. For example, the addition or removal of capacity, the deployment or rollback of updated software or configuration changes, or the migration of system data must be conducted to the desired availability goal. Compounding the costs for software development, at very high levels of availability, innovation suffers because of the need to move more slowly in deploying systems. The guidance, therefore, is to be thorough in applying the standards and considering the appropriate availability target for the entire lifecycle of operating the system.

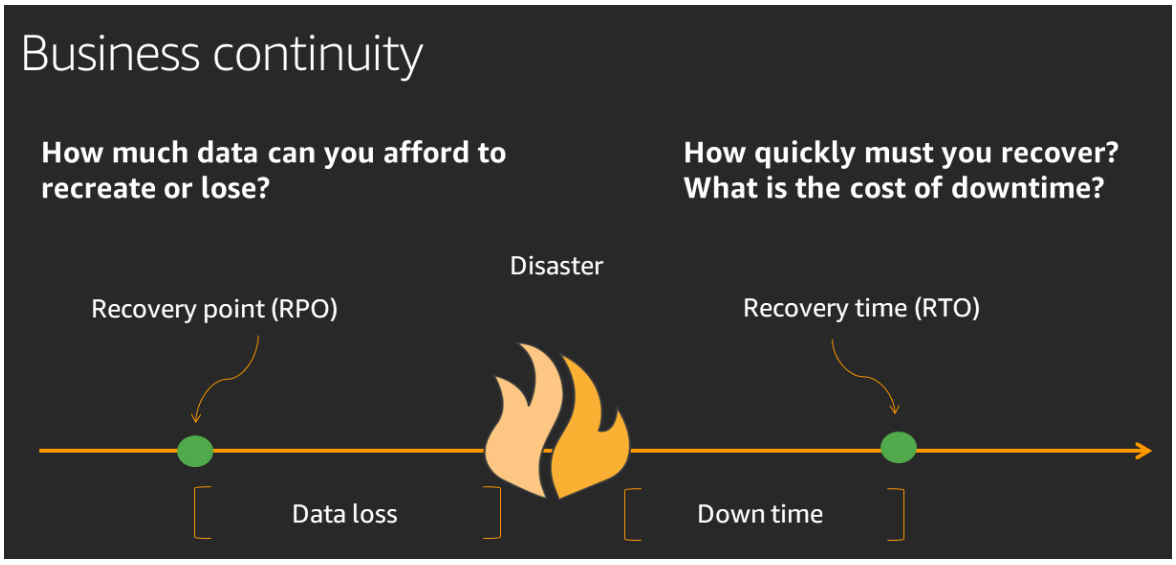
Another way that costs escalate in systems that operate with higher availability design goals is in the selection of dependencies. At these higher goals, the set of software or services that can be chosen as dependencies diminishes based on which of these services have had the deep investments we previously described. As the availability design goal increases, it's typical to find fewer multi-purpose services (such as a relational database) and more purpose-built services. This is because the latter are easier to evaluate, test, and automate, and have a reduced potential for surprise interactions with included but unused functionality.

Disaster Recovery (DR) Objectives

In addition to availability objectives, your resiliency strategy should also include Disaster Recovery (DR) objectives based on strategies to recover your workload in case of a disaster event. Disaster Recovery focuses on one-time recovery objectives in response natural disasters, large-scale technical failures, or human threats such as attack or error. This is different than availability which measures mean resiliency over a period of time in response to component failures, load spikes, or software bugs.

Recovery Time Objective (RTO) Defined by the organization. RTO is the maximum acceptable delay between the interruption of service and restoration of service. This determines what is considered an acceptable time window when service is unavailable.

Recovery Point Objective (RPO) Defined by the organization. RPO is the maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.



The relationship of RPO (Recovery Point Objective), RTO (Recovery Time Objective), and the disaster event.

RTO is similar to MTTR (Mean Time to Recovery) in that both measure the time between the start of an outage and workload recovery. However MTTR is a mean value taken over several availability impacting events over a period of time, while RTO is a target, or maximum value allowed, for a *single* availability impacting event.

Understanding Availability Needs

It's common to initially think of an application's availability as a single target for the application as a whole. However, upon closer inspection, we frequently find that certain aspects of an application or service have different availability requirements. For example, some systems might prioritize the ability to receive and store new data ahead of retrieving existing data. Other systems prioritize real-time operations over operations that change a system's configuration or environment. Services might have very high availability requirements during certain hours of the day, but can tolerate much longer periods of disruption outside of these hours. These are a few of the ways that you can decompose a single application into constituent parts, and evaluate the availability requirements for each. The benefit of doing this is to focus your efforts (and expense) on availability according to specific needs, rather than engineering the whole system to the strictest requirement.

Recommendation
Critically evaluate the unique aspects to your applications and, where appropriate, differentiate the availability and disaster recovery design goals to reflect the needs of your business.

Within AWS, we commonly divide services into the "data plane" and the "control plane." The data plane is responsible for delivering real-time service while control planes are used to configure the environment. For example, Amazon EC2 instances, Amazon RDS databases, and Amazon DynamoDB table read/write operations are all data plane operations. In contrast, launching new EC2 instances or RDS databases, or adding or changing table metadata in DynamoDB are all considered control plane operations. While high levels of availability are important for all of these capabilities, the data planes typically have higher availability design goals than the control planes. Therefore workloads with high availability requirements should avoid run-time dependency on control plane operations.

Many AWS customers take a similar approach to critically evaluating their applications and identifying subcomponents with different availability needs. Availability design goals are then tailored to the different aspects, and the appropriate work efforts are executed to engineer the system. AWS has significant experience engineering applications with a range of availability design goals, including services with 99.999% or greater availability. AWS Solution Architects (SAs) can help you design appropriately for your availability goals. Involving AWS early in your design process improves our ability to help you meet your availability goals. Planning for availability is not only done before your workload launches. It's also done continuously to refine your design as you gain operational experience, learn from real world events, and endure failures of different types. You can then apply the appropriate work effort to improve upon your implementation.

The availability needs that are required for a workload must be aligned to the business need and criticality. By first defining business criticality framework with defined RTO, RPO, and availability, you can then assess each workload. Such an approach requires that the people involved in implementation of the workload are knowledgeable of the framework, and the impact their workload has on business needs.

Foundations

Foundational requirements are those whose scope extends beyond a single workload or project. Before architecting any system, foundational requirements that influence reliability should be in place. For example, you must have sufficient network bandwidth to your data center.

In an on-premises environment, these requirements can cause long lead times due to dependencies and therefore must be incorporated during initial planning. With AWS however, most of these foundational requirements are already incorporated or can be addressed as needed. The cloud is designed to be nearly limitless, so it's the responsibility of AWS to satisfy the requirement for sufficient networking and compute capacity, leaving you free to change resource size and allocations on demand.

The following sections explain best practices that focus on these considerations for reliability.

Topics

- Manage Service Quotas and Constraints (p. 8)
- Plan your Network Topology (p. 9)

Manage Service Quotas and Constraints

For cloud-based workload architectures, there are service quotas (which are also referred to as service limits). These quotas exist to prevent accidentally provisioning more resources than you need and to limit request rates on API operations so as to protect services from abuse. There are also resource constraints, for example, the rate that you can push bits down a fiber-optic cable, or the amount of storage on a physical disk.

If you are using AWS Marketplace applications, you must understand the limitations of those applications. If you are using third-party web services or software as a service, you must be aware of those limits also.

Aware of service quotas and constraints: You are aware of your default quotas and quota increase requests for your workload architecture. You additionally know which resource constraints, such as disk or network, are potentially impactful.

Service Quotas is an AWS service that helps you manage your quotas for over 100 AWS services from one location. Along with looking up the quota values, you can also request and track quota increases from the Service Quotas console or via the AWS SDK. AWS Trusted Advisor offers a service quotas check that displays your usage and quotas for some aspects of some services. The default service quotas per service are also in the AWS documentation per respective service, for example, see Amazon VPC Quotas. Rate limits on throttled APIs are set within the API Gateway itself by configuring a usage plan. Other limits that are set as configuration on their respective services include Provisioned IOPS, RDS storage allocated, and EBS volume allocations. Amazon Elastic Compute Cloud (Amazon EC2) has its own service limits dashboard that can help you manage your instance, Amazon Elastic Block Store (Amazon EBS), and Elastic IP address limits. If you have a use case where service quotas impact your application's performance and they are not adjustable to your needs, then contact AWS Support to see if there are mitigations.

Manage quotas across accounts and regions: If you are using multiple AWS accounts or AWS Regions, ensure that you request the appropriate quotas in all environments in which your production workloads run.

Service quotas are tracked per account. Unless otherwise noted, each quota is AWS Region-specific.

In addition to the production environments, also manage quotas in all applicable non-production environments, so that testing and development are not hindered.

Accommodate fixed service quotas and constraints through architecture: Be aware of unchangeable service quotas and physical resources, and architect to prevent these from impacting reliability.

Examples include network bandwidth, AWS Lambda payload size, throttle burst rate for API Gateway, and concurrent user connections to an Amazon Redshift cluster.

Monitor and manage quotas: Evaluate your potential usage and increase your quotas appropriately allowing for planned growth in usage.

For supported services, you can manage your quotas by configuring CloudWatch alarms to monitor usage and alert you to approaching quotas. These alarms can be triggered from Service Quotas or from Trusted Advisor. You can also use metric filters on CloudWatch Logs to search and extract patterns in logs to determine if usage is approaching quota thresholds.

Automate quota management: Implement tools to alert you when thresholds are being approached. By using Service Quotas APIs, you can automate quota increase requests.

If you integrate your Configuration Management Database (CMDB) or ticketing system with Service Quotas, you can automate the tracking of quota increase requests and current quotas. In addition to the AWS SDK, Service Quotas offers automation using AWS command line tools.

Ensure that a sufficient gap exists between the current quotas and the maximum usage to accommodate failover: When a resource fails, it may still be counted against quotas until it's successfully terminated. Ensure that your quotas cover the overlap of all failed resources with replacements before the failed resources are terminated. You should consider an Availability Zone failure when calculating this gap.

Resources

Video

- AWS Live re:Inforce 2019 - Service Quotas

Documentation

- What Is Service Quotas?
- AWS Service Quotas (formerly referred to as service limits)
- Amazon EC2 Service Limits
- AWS Trusted Advisor Best Practice Checks (see the **Service Limits** section)
- AWS Limit Monitor on AWS Answers
- AWS Marketplace: CMDB products that help track limits
- APN Partner: partners that can help with configuration management

Plan your Network Topology

Workloads often exist in multiple environments. These include multiple cloud environments (both publicly accessible and private) and possibly your existing data center infrastructure. Plans must

include network considerations, such as intrasystem and intersystem connectivity, public IP address management, private IP address management, and domain name resolution.

When architecting systems using IP address-based networks, you must plan network topology and addressing in anticipation of possible failures, and to accommodate future growth and integration with other systems and their networks.

Amazon Virtual Private Cloud (Amazon VPC) lets you provision a private, isolated section of the AWS Cloud where you can launch AWS resources in a virtual network.

Use highly available network connectivity for your workload public endpoints: These endpoints and the routing to them must be highly available. To achieve this, use highly available DNS, content delivery networks (CDNs), API Gateway, load balancing, or reverse proxies.

Amazon Route 53, AWS Global Accelerator, Amazon CloudFront, Amazon API Gateway, and Elastic Load Balancing (ELB) all provide highly available public endpoints. You might also choose to evaluate AWS Marketplace software appliances for load balancing and proxying.

Consumers of the service your workload provides, whether they are end-users or other services, make requests on these service endpoints. Several AWS resources are available to enable you to provide highly available endpoints.

Elastic Load Balancing provides load balancing across Availability Zones, performs Layer 4 (TCP) or Layer 7 (http/https) routing, integrates with AWS WAF, and integrates with AWS Auto Scaling to help create a self-healing infrastructure and absorb increases in traffic while releasing resources when traffic decreases.

Amazon Route 53 is a scalable and highly available Domain Name System (DNS) service that connects user requests to infrastructure running in AWS-such as Amazon EC2 instances, Elastic Load Balancing load balancers, or Amazon S3 buckets–and can also be used to route users to infrastructure outside of AWS.

AWS Global Accelerator is a network layer service that you can use to direct traffic to optimal endpoints over the AWS global network.

Distributed Denial of Service (DDoS) attacks risk shutting out legitimate traffic and lowering availability for your users. AWS Shield provides automatic protection against these attacks at no extra cost for AWS service endpoints on your workload. You can augment these features with virtual appliances from APN Partners and the AWS Marketplace to meet your needs.

Provision redundant connectivity between private networks in the cloud and on-premises environments: Use multiple AWS Direct Connect (DX) connections or VPN tunnels between separately deployed private networks. Use multiple DX locations for high availability. If using multiple AWS Regions, ensure redundancy in at least two of them. You might want to evaluate AWS Marketplace appliances that terminate VPNs. If you use AWS Marketplace appliances, deploy redundant instances for high availability in different Availability Zones.

AWS Direct Connect is a cloud service that makes it easy to establish a dedicated network connection from your on-premises environment to AWS. Using Direct Connect Gateway, your on-premises data center can be connected to multiple AWS VPCs spread across multiple AWS Regions.

This redundancy addresses possible failures that impact connectivity resiliency:

- How are you going to be resilient to failures in your topology?
- What happens if you misconfigure something and remove connectivity?
- Will you be able to handle an unexpected increase in traffic/use of your services?
- Will you be able to absorb an attempted Distributed Denial of Service (DDoS) attack?

When connecting your VPC to your on-premises data center via VPN, you should consider the resiliency and bandwidth requirements that you need when you select the vendor and instance size on which you need to run the appliance. If you use a VPN appliance that is not resilient in its implementation, then you should have a redundant connection through a second appliance. For all these scenarios, you need to define an acceptable time to recovery and test to ensure that you can meet those requirements.

If you choose to connect your VPC to your data center using a Direct Connect connection and you need this connection to be highly available, have redundant DX connections from each data center. The redundant connection should use a second DX connection from different location than the first. If you have multiple data centers, ensure that the connections terminate at different locations. Use the Direct Connect Resiliency Toolkit to help you set this up.

If you choose to fail over to VPN over the internet using AWS VPN, it's important to understand that it supports up to 1.25-Gbps throughput per VPN tunnel, but does not support Equal Cost Multi Path (ECMP) for outbound traffic in the case of multiple AWS Managed VPN tunnels terminating on the same VGW. We do not recommend that you use AWS Managed VPN as a backup for Direct Connect connections unless you can tolerate speeds less than 1 Gbps during failover.

You can also use VPC endpoints to privately connect your VPC to supported AWS services and VPC endpoint services powered by AWS PrivateLink without traversing the public internet. Endpoints are virtual devices. They are horizontally scaled, redundant, and highly available VPC components. They allow communication between instances in your VPC and services without imposing availability risks or bandwidth constraints on your network traffic.

Ensure IP subnet allocation accounts for expansion and availability: Amazon VPC IP address ranges must be large enough to accommodate workload requirements, including factoring in future expansion and allocation of IP addresses to subnets across Availability Zones. This includes load balancers, EC2 instances, and container-based applications.

When you plan your network topology, the first step is to define the IP address space itself. Private IP address ranges (following RFC 1918 guidelines) should be allocated for each VPC. Accommodate the following requirements as part of this process:

- Allow IP address space for more than one VPC per Region.
- Within a VPC, allow space for multiple subnets that span multiple Availability Zones.
- Always leave unused CIDR block space within a VPC for future expansion.
- Ensure that there is IP address space to meet the needs of any transient fleets of EC2 instances that you might use, such as Spot Fleets for machine learning, Amazon EMR clusters, or Amazon Redshift clusters.
- Note that the first four IP addresses and the last IP address in each subnet CIDR block are reserved and not available for your use.

  You should plan on deploying large VPC CIDR blocks. Note that the initial VPC CIDR block allocated to your VPC cannot be changed or deleted, but you can add additional non-overlapping CIDR blocks to the VPC. Subnet IPv4 CIDRs cannot be changed, however IPv6 CIDRs can. Keep in mind that deploying the largest VPC possible (/16) results in over 65,000 IP addresses. In the base 10.x.x.x IP address space alone, you could provision 255 such VPCs. You should therefore err on the side of being too large rather than too small to make it easier to manage your VPCs.

Prefer hub-and-spoke topologies over many-to-many mesh: If more than two network address spaces (for example, VPCs and on-premises networks) are connected via VPC peering, AWS Direct Connect, or VPN, then use a hub-and-spoke model, like those provided by AWS Transit Gateway.

If you have only two such networks, you can simply connect them to each other, but as the number of networks grows, the complexity of such meshed connections becomes untenable. AWS Transit Gateway provides an easy to maintain hub-and-spoke model, allowing the routing of traffic across your multiple networks.

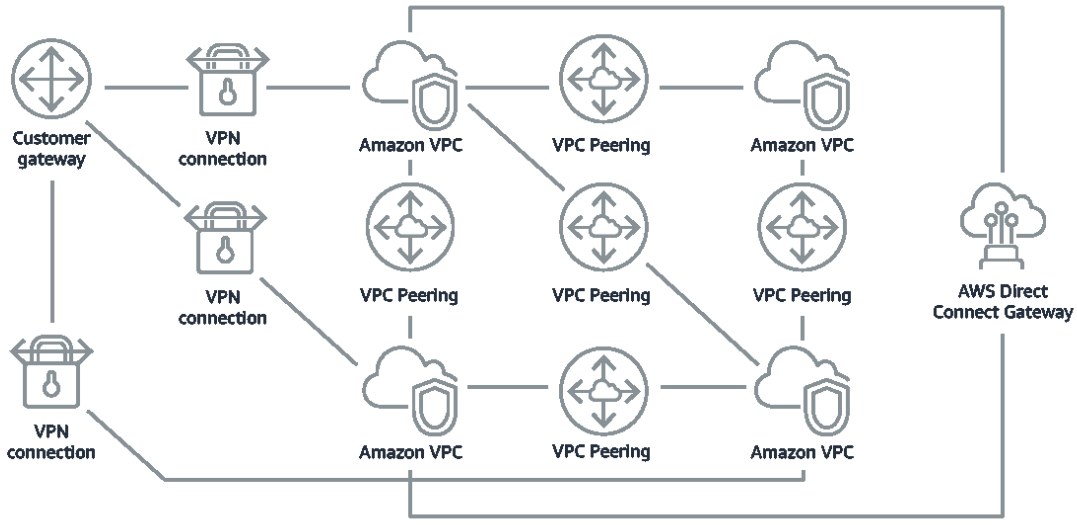


Figure 1: Without AWS Transit Gateway: You need to peer each Amazon VPC to each other and to each onsite location using a VPN connection, which can become complex as it scales.

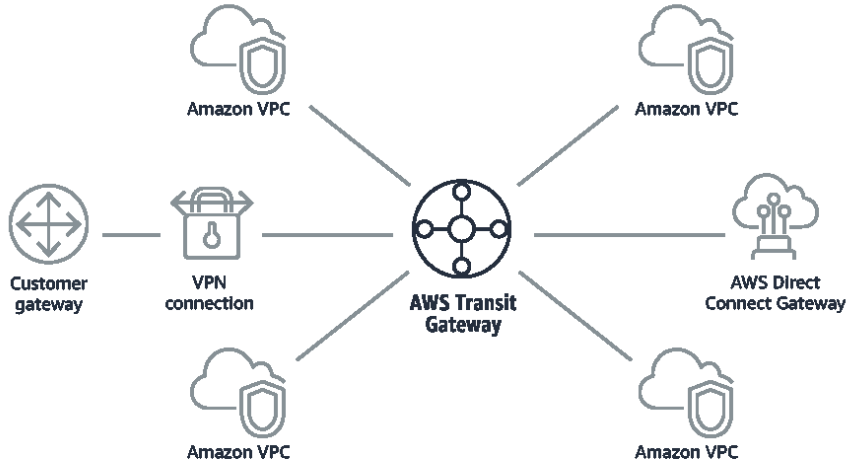


Figure 2: With AWS Transit Gateway: You simply connect each Amazon VPC or VPN to the AWS Transit Gateway and it routes traffic to and from each VPC or VPN.

Enforce non-overlapping private IP address ranges in all private address spaces where they are connected: The IP address ranges of each of your VPCs must not overlap when peered or connected via VPN. You must similarly avoid IP address conflicts between a VPC and on-premises environments or with other cloud providers that you use. You must also have a way to allocate private IP address ranges when needed.

An IP address management (IPAM) system can help with this. Several IPAMs are available from the AWS Marketplace.

Resources

Videos

- AWS re:Invent 2018: Advanced VPC Design and New Capabilities for Amazon VPC (NET303)
- AWS re:Invent 2019: AWS Transit Gateway reference architectures for many VPCs (NET406-R1)

Documentation

- What Is a Transit Gateway?
- What Is Amazon VPC?
- Working with Direct Connect Gateways
- Using the Direct Connect Resiliency Toolkit to get started
- AWS Direct Connect Resiliency Recommendations
- Multiple data center HA network connectivity
- What Is AWS Global Accelerator?
- Using redundant Site-to-Site VPN connections to provide failover
- VPC Endpoints and VPC Endpoint Services (AWS PrivateLink)
- Amazon Virtual Private Cloud Connectivity Options Whitepaper
- AWS Marketplace for Network Infrastructure
- APN Partner: partners that can help plan your networking

Workload Architecture

A reliable workload starts with upfront design decisions for both software and infrastructure. Your architecture choices will impact your workload behavior across all five Well-Architected pillars. For reliability, there are specific patterns you must follow.

The following sections explain best practices to use with these patterns for reliability.

Topics

- Design Your Workload Service Architecture (p. 14)
- Design Interactions in a Distributed System to Prevent Failures (p. 16)
- Design Interactions in a Distributed System to Mitigate or Withstand Failures (p. 19)

Design Your Workload Service Architecture

Build highly scalable and reliable workloads using a service-oriented architecture (SOA) or a microservices architecture. Service-oriented architecture (SOA) is the practice of making software components reusable via service interfaces. Microservices architecture goes further to make components smaller and simpler.

Service-oriented architecture (SOA) interfaces use common communication standards so that they can be rapidly incorporated into new workloads. SOA replaced the practice of building monolith architectures, which consisted of interdependent, indivisible units.

At AWS, we have always used SOA, but have now embraced building our systems using microservices. While microservices have several attractive qualities, the most important benefit for availability is that microservices are smaller and simpler. They allow you to differentiate the availability required of different services, and thereby focus investments more specifically to the microservices that have the greatest availability needs. For example, to deliver product information pages on Amazon.com (“detail pages”), hundreds of microservices are invoked to build discrete portions of the page. While there are a few services that must be available to provide the price and the product details, the vast majority of content on the page can simply be excluded if the service isn’t available. Even such things as photos and reviews are not required to provide an experience where a customer can buy a product.

Choose how to segment your workload: Monolithic architecture should be avoided. Instead, you should choose between SOA and microservices. When making each choice, balance the benefits against the complexities—what is right for a new product racing to first launch is different than what a workload built to scale from the start needs. The benefits of using smaller segments include greater agility, organizational flexibility, and scalability. Complexities include possible increased latency, more complex debugging, and increased operational burden.

Even if you choose to start with a monolith architecture, you must ensure that it’s modular and has the ability to ultimately evolve to SOA or microservices as your product scales with user adoption. SOA and microservices offer respectively smaller segmentation, which is preferred as a modern scalable and reliable architecture, but there are trade-offs to consider especially, when deploying a microservice architecture. One is that you now have a distributed compute architecture that can make it harder to achieve user latency requirements and there is additional complexity in debugging and tracing of user interactions. AWS X-Ray can be used to assist you in solving this problem. Another effect to consider is increased operational complexity as you proliferate the number of applications that you are managing, which requires the deployment of multiple independency components.

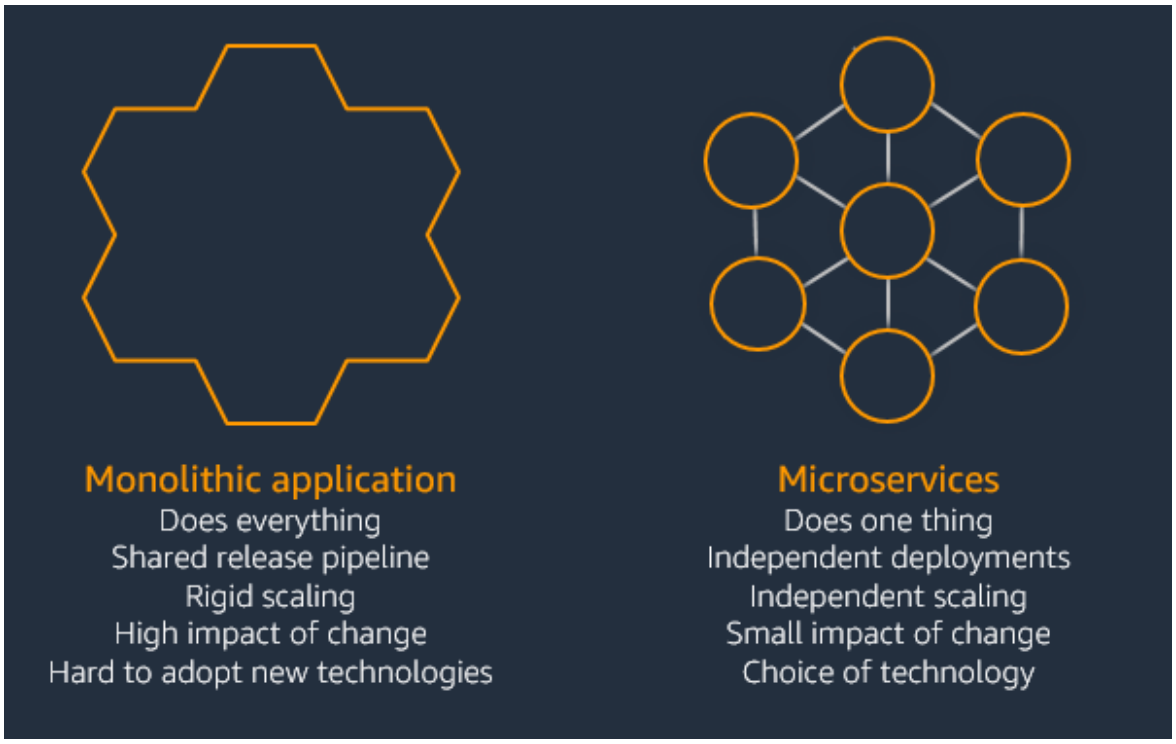
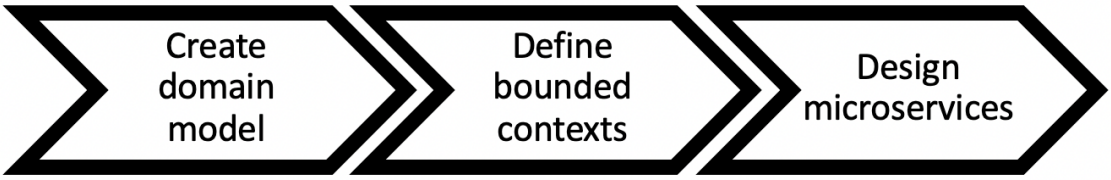


Figure 3: Monolithic architecture versus microservices architecture

Build services focused on specific business domains and functionality: SOA builds services with well-delineated functions defined by business needs. Microservices use domain models and bounded context to limit this further so that each service does just *one thing*. Focusing on specific functionality enables you to differentiate the reliability requirements of different services, and target investments more specifically. A concise business problem and small team associated with each service also enables easier organizational scaling.

In designing a microservice architecture, it's helpful to use Domain-Driven Design (DDD) to model the business problem using entities. For example for Amazon.com entities may include package, delivery, schedule, price, discount, and currency. Then the model is further divided into smaller models using Bounded Context, where entities that share similar features and attributes are grouped together. So using the Amazon example package, delivery and schedule would be part of the shipping context, while price, discount, and currency are part of the pricing context. With the model divided into contexts, a template for how to boundary microservices emerges.



Provide service contracts per API: Service contracts are documented agreements between teams on service integration and include a machine-readable API definition, rate limits, and performance expectations. A versioning strategy allows clients to continue using the existing API and migrate their applications to the newer API when they are ready. Deployment can happen anytime, as long as the contract is not violated. The service provider team can use the technology stack of their choice to satisfy the API contract. Similarly, the service consumer can use their own technology.

Microservices take the concept of SOA to the point of creating services that have a minimal set of functionality. Each service publishes an API and design goals, limits, and other considerations for using the service. This establishes a "contract" with calling applications. This accomplishes three main benefits:

- The service has a concise business problem to be served and a small team that owns the business problem. This allows for better organizational scaling.
- The team can deploy at any time as long as they meet their API and other "contract" requirements.
- The team can use any technology stack they want to as long as they meet their API and other "contract" requirements.

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. It handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, authorization and access control, monitoring, and API version management. Using OpenAPI Specification (OAS), formerly known as the Swagger Specification, you can define your API contract and import it into API Gateway. With API Gateway, you can then version and deploy the APIs.

Resources

Documentation

- **Amazon API Gateway:** Configuring a REST API Using OpenAPI
- Implementing Microservices on AWS
- Microservices on AWS

External Links

- **Microservices - a definition of this new architectural term**
- Microservice Trade-Offs
- Bounded Context (a central pattern in Domain-Driven Design)

Design Interactions in a Distributed System to Prevent Failures

Distributed systems rely on communications networks to interconnect components, such as servers or services. Your workload must operate reliably despite data loss or latency in these networks. Components of the distributed system must operate in a way that does not negatively impact other components or the workload. These best practices prevent failures and improve mean time between failures (MTBF).

Identify which kind of distributed system is required: Hard real-time distributed systems require responses to be given synchronously and rapidly, while soft real-time systems have a more generous time window of minutes or more for response. Offline systems handle responses through batch or asynchronous processing. Hard real-time distributed systems have the most stringent reliability requirements.

The most difficult challenges with distributed systems are for the hard real-time distributed systems, also known as request/reply services. What makes them difficult is that requests arrive unpredictably and responses must be given rapidly (for example, the customer is actively waiting for the response). Examples include front-end web servers, the order pipeline, credit card transactions, every AWS API, and telephony.

Implement loosely coupled dependencies: Dependencies such as queuing systems, streaming systems, workflows, and load balancers are loosely coupled. Loose coupling helps isolate behavior of a component from other components that depend on it, increasing resiliency and agility.

If changes to one component force other components that rely on it to also change, then they are *tightly* coupled. *Loose* coupling breaks this dependency so that dependent components only need to know the versioned and published interface. Implementing loose coupling between dependencies isolates a failure in one from impacting another.

Loose coupling enables you to add additional code or features to a component while minimizing risk to components that depend on it. Also scalability is improved as you can scale out or even change underlying implementation of the dependency.

To further improve resiliency through loose coupling, make component interactions asynchronous where possible. This model is suitable for any interaction that does not need an immediate response and where an acknowledgment that a request has been registered will suffice. It involves one component that generates events and another that consumes them. The two components do not integrate through direct point-to-point interaction but usually through an intermediate durable storage layer, such as an SQS queue or a streaming data platform such as Amazon Kinesis, or AWS Step Functions.

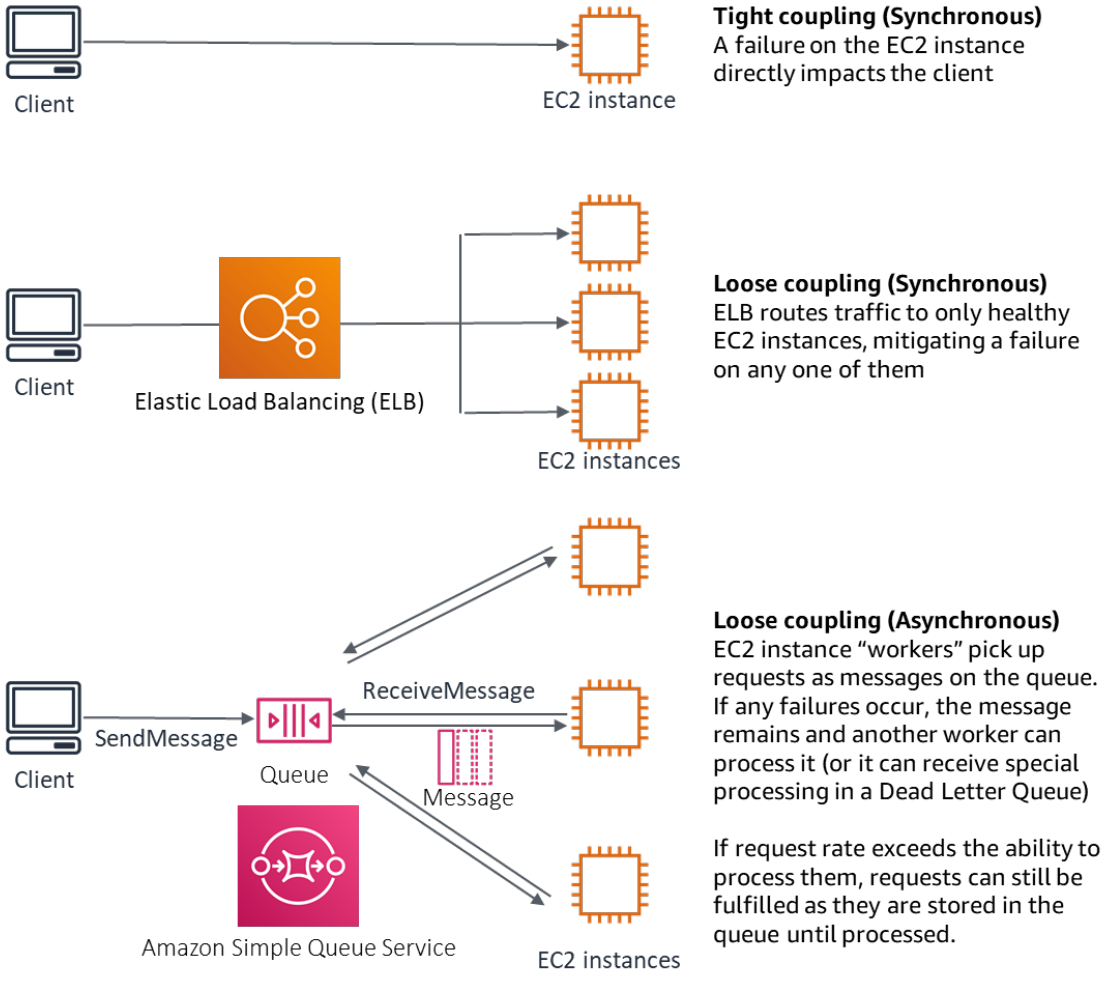


Figure 4: Dependencies such as queuing systems and load balancers are loosely coupled

Amazon SQS queues and Elastic Load Balancers are just two ways to add an intermediate layer for loose coupling. Event-driven architectures can also be built in the AWS Cloud using Amazon EventBridge,

which can abstract clients (event producers) from the services they rely on (event consumers). Amazon Simple Notification Service is an effective solution when you need high-throughput, push-based, many-to-many messaging. Using Amazon SNS topics, your publisher systems can fan out messages to a large number of subscriber endpoints for parallel processing.

While queues offer several advantages, in most hard real-time systems, requests older than a threshold time (often seconds) should be considered stale (the client has given up and is no longer waiting for a response), and not processed. This way more recent (and likely still valid requests) can be processed instead.

Make all responses idempotent: An idempotent service promises that each request is completed *exactly once*, such that making multiple identical requests has the same effect as making a single request. An idempotent service makes it easier for a client to implement retries without fear that a request will be erroneously processed multiple times. To do this, clients can issue API requests with an idempotency token—the same token is used whenever the request is repeated. An idempotent service API uses the token to return a response identical to the response that was returned the first time that the request was completed.

In a distributed system, it's easy to perform an action at most once (client makes only one request), or at least once (keep requesting until client gets confirmation of success). But it's hard to guarantee an action is idempotent, which means it's performed *exactly* once, such that making multiple identical requests has the same effect as making a single request. Using idempotency tokens in APIs, services can receive a mutating request one or more times without creating duplicate records or side effects.

Do constant work: Systems can fail when there are large, rapid changes in load. For example, if your workload is doing a health check that monitors the health of thousands of servers, it should send the same size payload (a full snapshot of the current state) each time. Whether no servers are failing, or all of them, the health check system is doing constant work with no large, rapid changes.

For example, if the health check system is monitoring 100,000 servers, the load on it is nominal under the normally light server failure rate. However, if a major event makes half of those servers unhealthy, then the health check system would be overwhelmed trying to update notification systems and communicate state to its clients. So instead the health check system should send the full snapshot of the current state each time. 100,000 server health states, each represented by a bit, would only be a 12.5-KB payload. Whether no servers are failing, or all of them are, the health check system is doing constant work, and large, rapid changes are not a threat to the system stability. This is actually how Amazon Route 53 handles health checks for endpoints (such as IP addresses) to determine how end users are routed to them.

Resources

Videos

- AWS re:Invent 2019: Moving to event-driven architectures (SVS308)
- AWS re:Invent 2018: Close Loops & Opening Minds: How to Take Control of Systems, Big & Small ARC337 (includes loose coupling, constant work, static stability)
- AWS New York Summit 2019: Intro to Event-driven Architectures and Amazon EventBridge (MAD205) (discusses EventBridge, Amazon SQS, Amazon SNS)

Documentation

- AWS Services That Publish CloudWatch Metrics
- What Is Amazon Simple Queue Service?
- Amazon EC2: Ensuring Idempotency
- The Amazon Builders' Library: Challenges with distributed systems

- The Amazon Builders' Library: Reliability, constant work, and a good cup of coffee
- Centralized Logging solution
- AWS Marketplace: products that can be used for monitoring and alerting
- APN Partner: partners that can help you with monitoring and logging

Design Interactions in a Distributed System to Mitigate or Withstand Failures

Distributed systems rely on communications networks to interconnect components (such as servers or services). Your workload must operate reliably despite data loss or latency over these networks. Components of the distributed system must operate in a way that does not negatively impact other components or the workload. These best practices enable workloads to withstand stresses or failures, more quickly recover from them, and mitigate the impact of such impairments. The result is improved mean time to recovery (MTTR).

These best practices prevent failures and improve mean time between failures (MTBF).

Implement graceful degradation to transform applicable hard dependencies into soft dependencies: When a component's dependencies are unhealthy, the component itself can still function, although in a degraded manner. For example, when a dependency call fails, instead use a predetermined static response.

Consider a service B that is called by service A and in turn calls service C.

Figure 5: Service C fails when called from service B. Service B returns a degraded response to service A.

When service B calls service C, it received an error or timeout from it. Service B, lacking a response from service C (and the data it contains) instead returns what it can. This can be the last cached good value, or service B can substitute a pre-determined static response for what it would have received from service C. It can then return a degraded response to its caller, service A. Without this static response, the failure in service C would cascade through service B to service A, resulting in a loss of availability.

As per the multiplicative factor in the availability equation for hard dependencies (see Calculating availability with hard dependencies (p. 4)), any drop in the availability of C seriously impacts effective availability of B. By returning the static response service B mitigates the failure in C and, although degraded, makes service C's availability look like 100% availability (assuming it reliably returns the static response under error conditions). Note that the static response is a simple alternative to returning an error, and is not an attempt to re-compute the response using different means. Such attempts at a completely different mechanism to try to achieve the same result are called fallback behavior, and are an anti-pattern to be avoided.

Another example of graceful degradation is the *circuit breaker pattern*. Retry strategies should be used when the failure is transient. When this is not the case, and the operation is likely to fail, the circuit breaker pattern prevents the client from performing a request that is likely to fail. When requests are being processed normally, the circuit breaker is closed and requests flow through. When the remote system begins returning errors or exhibits high latency, the circuit breaker opens and the dependency is ignored or results are replaced with more simply obtained but less comprehensive responses (which

might simply be a response cache). Periodically, the system attempts to call the dependency to determine if it has recovered. When that occurs, the circuit breaker is closed.

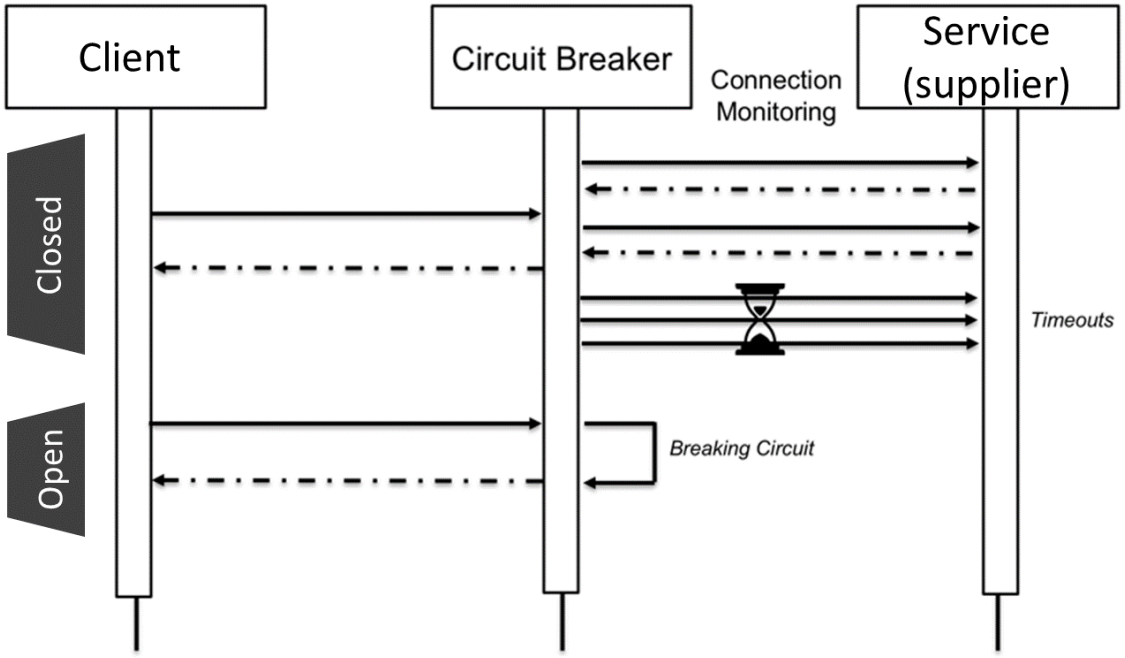


Figure 6: Circuit breaker showing closed and open states.

In addition to the closed and open states shown in the diagram, after a configurable period of time in the open state, the circuit breaker can transition to half-open. In this state, it periodically attempts to call the service at a much lower rate than normal. This probe is used to check the health of the service. After a number of successes in half-open state, the circuit breaker transitions to closed, and normal requests resume.

Throttle requests: This is a mitigation pattern to respond to an unexpected increase in demand. Some requests are honored but those over a defined limit are rejected and return a message indicating they have been throttled. The expectation on clients is that they will back off and abandon the request or try again at a slower rate.

Your services should be designed to a known capacity of requests that each node or cell can process. This can be established through load testing. You then need to track the arrival rate of requests and if the temporary arrival rate exceeds this limit, the appropriate response is to signal that the request has been throttled. This allows the user to retry, potentially to a different node/cell that might have available capacity. Amazon API Gateway provides methods for throttling requests. Amazon SQS and Amazon Kinesis can buffer requests, smoothing out request rate and alleviate the need for throttling for requests that can be addressed asynchronously.

Control and limit retry calls: Use exponential backoff to retry after progressively longer intervals. Introduce jitter to randomize those retry intervals, and limit the maximum number of retries.

Typical components in a distributed software system include servers, load balancers, databases, and DNS servers. In operation, and subject to failures, any of these can start generating errors. The default technique for dealing with errors is to implement retries on the client side. This technique increases the reliability and availability of the application. However, at scale—and if clients attempt to retry the failed operation as soon as an error occurs—the network can quickly become saturated with new and retried requests, each competing for network bandwidth. This can result in a *retry storm,* which will reduce availability of the service. This pattern might continue until a full system failure occurs.

To avoid such scenarios, backoff algorithms such as the common **exponential backoff** should be used. Exponential backoff algorithms gradually decrease the rate at which retries are performed, thus avoiding network congestion.

Many SDKs and software libraries, including those from AWS, implement a version of these algorithms. However, **never assume a backoff algorithm exists—always test and verify this to be the case.**

Simple backoff alone is not enough because in distributed systems all clients may backoff simultaneously, creating clusters of retry calls. Marc Brooker in his blog post Exponential Backoff And Jitter, explains how to modify the `wait()` function in the exponential backoff to prevent clusters of retry calls. The solution is to add **jitter** in the `wait()` function. To avoid retrying for too long, implementations should cap the backoff to a maximum value.

Finally, it's important to configure a **maximum number of retries** or elapsed time, after which retrying will simply fail. AWS SDKs implement this by default, and it can be configured. For services lower in the stack, a maximum retry limit of zero or one will limit risk yet still be effective as retries are delegated to services higher in the stack.

Fail fast and limit queues: If the workload is unable to respond successfully to a request, then fail fast. This allows the releasing of resources associated with a request, and permits the service to recover if it's running out of resources. If the workload is able to respond successfully but the rate of requests is too high, then use a queue to buffer requests instead. However, do not allow long queues that can result in serving stale requests that the client has already given up on.

This best practice applies to the server-side, or receiver, of the request.

Be aware that queues can be created at multiple levels of a system, and can seriously impede the ability to quickly recover as older stale requests (that no longer need a response) are processed before newer requests in need of a response. Be aware of places where queues exist. They often hide in workflows or in work that's recorded to a database.

Set client timeouts: Set timeouts appropriately, verify them systematically, and do not rely on default values as they are generally set too high.

This best practice applies to the client-side, or sender, of the request.

Set both a connection timeout and a request timeout on any remote call, and generally on any call across processes. Many frameworks offer built-in timeout capabilities, but be careful as many have default values that are infinite or too high. A value that is too high reduces the usefulness of the timeout because resources continue to be consumed while the client waits for the timeout to occur. A too low value can generate increased traffic on the backend and increased latency because too many requests are retried. In some cases, this can lead to complete outages because all requests are being retried.

To learn more about how Amazon use timeouts, retries, and backoff with jitter, refer to the Builder's Library: Timeouts, retries, and backoff with jitter.

Make services stateless where possible: Services should either not require state, or should offload state such that between different client requests, there is no dependence on locally stored data on disk or in memory. This enables servers to be replaced at will without causing an availability impact. Amazon ElastiCache or Amazon DynamoDB are good destinations for offloaded state.

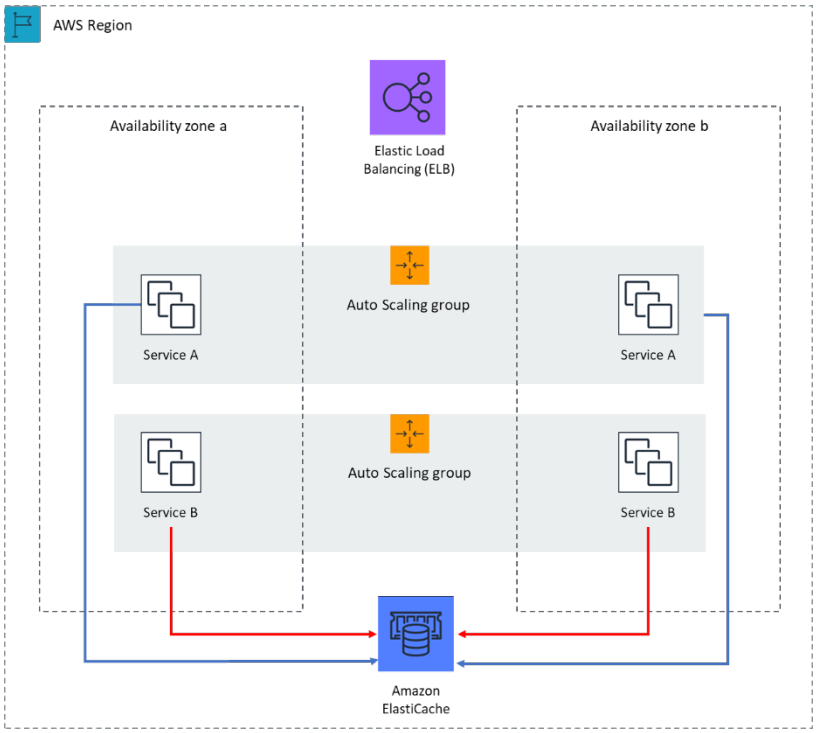


Figure 7: In this stateless web application, session state is offloaded to Amazon ElastiCache.

When users or services interact with an application, they often perform a series of interactions that form a session. A session is unique data for users that persists between requests while they use the application. A stateless application is an application that does not need knowledge of previous interactions and does not store session information.

Once designed to be stateless, you can then use serverless compute platforms, such as AWS Lambda or AWS Fargate.

In addition to server replacement, another benefit of stateless applications is that they can scale horizontally because any of the available compute resources (such as EC2 instances and AWS Lambda functions) can service any request.

Implement emergency levers: These are rapid processes that may mitigate availability impact on your workload. They can be operated in the absence of a root cause. An ideal emergency lever reduces the cognitive burden on the resolvers to zero by providing fully deterministic activation and deactivation criteria. Example levers include blocking all robot traffic or serving a static response. Levers are often manual, but they can also be automated.

Tips for implementing and using emergency levers:

- When levers are activated, do LESS, not more
- Keep it simple, avoid bimodal behavior
- Test your levers periodically

These are examples of actions that are NOT emergency levers:

- Add capacity
- Call up service owners of clients that depend on your service and ask them to reduce calls
- Making a change to code and releasing it

Resources

Video

- Retry, backoff, and jitter: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)

Documentation

- Error Retries and Exponential Backoff in AWS
- Amazon API Gateway: Throttle API Requests for Better Throughput
- The Amazon Builders' Library: Timeouts, retries, and backoff with jitter
- The Amazon Builders' Library: Avoiding fallback in distributed systems
- The Amazon Builders' Library: Avoiding insurmountable queue backlogs
- The Amazon Builders' Library: Caching challenges and strategies

Labs

- Well-Architected lab: Level 300: Implementing Health Checks and Managing Dependencies to Improve Reliability

External Links

- CircuitBreaker (summarizes Circuit Breaker from "Release It!" book)

Books

- Michael Nygard "Release It! Design and Deploy Production-Ready Software"

Change Management

Changes to your workload or its environment must be anticipated and accommodated to achieve reliable operation of the workload. Changes include those imposed on your workload such as spikes in demand, as well as those from within such as feature deployments and security patches.

The following sections explain the best practices for change management.

Topics

- Monitor Workload Resources (p. 24)
- Design your Workload to Adapt to Changes in Demand (p. 27)
- Implement Change (p. 29)

Monitor Workload Resources

Logs and metrics are powerful tools to gain insight into the health of your workload. You can configure your workload to monitor logs and metrics and send notifications when thresholds are crossed or significant events occur. Monitoring enables your workload to recognize when low-performance thresholds are crossed or failures occur, so it can recover automatically in response.

Monitoring is critical to ensure that you are meeting your availability requirements. Your monitoring needs to effectively detect failures. The worst failure mode is the “silent” failure, where the functionality is no longer working, but there is no way to detect it except indirectly. Your customers know before you do. Alerting when you have problems is one of the primary reasons you monitor. Your alerting should be decoupled from your systems as much as possible. If your service interruption removes your ability to alert, you will have a longer period of interruption.

At AWS, we instrument our applications at multiple levels. We record latency, error rates, and availability for each request, for all dependencies, and for key operations within the process. We record metrics of successful operation as well. This allows us to see impending problems before they happen. We don’t just consider average latency. We focus even more closely on latency outliers, like the 99.9th and 99.99th percentile. This is because if one request out of 1,000 or 10,000 is slow, that is still a poor experience. Also, although your average may be acceptable, if one in 100 of your requests causes extreme latency, it will eventually become a problem as your traffic grows.

Monitoring at AWS consists of four distinct phases:

1. Generation — Monitor all components for the workload
2. Aggregation — Define and calculate metrics
3. Real-time processing and alarming — Send notifications and automate responses
4. Storage and Analytics

Generation — Monitor all components for the workload: Monitor the components of the workload with Amazon CloudWatch or third-party tools. Monitor AWS services with AWS Personal Health Dashboard.

All components of your workload should be monitored, including the front-end, business logic, and storage tiers. Define key metrics and how to extract them from logs, if necessary, and set create thresholds for corresponding alarm events

Monitoring in the cloud offers new opportunities. Most cloud providers have developed customizable hooks and insights into multiple layers of your workload.

AWS makes an abundance of monitoring and log information available for consumption, which can be used to define change-in-demand processes. The following is just a partial list of services and features that generate log and metric data.

- Amazon ECS, Amazon EC2, Elastic Load Balancing, AWS Auto Scaling, and Amazon EMR publish metrics for CPU, network I/O, and disk I/O averages.
- Amazon CloudWatch Logs can be enabled for Amazon Simple Storage Service (Amazon S3), Classic Load Balancers, and Application Load Balancers.
- VPC Flow Logs can be enabled to analyze network traffic into and out of a VPC.
- AWS CloudTrail logs AWS account activity, including actions taken through the AWS Management Console, AWS SDKs, command line tools.
- Amazon EventBridge delivers a real-time stream of system events that describes changes in AWS services.
- AWS provides tooling to collect operating system-level logs and stream them into CloudWatch Logs.
- Custom Amazon CloudWatch metrics can be used for metrics of any dimension.
- Amazon ECS and AWS Lambda stream log data to CloudWatch Logs.
- AWS AI and ML services, such as Amazon Rekognition, Amazon Lex, and Amazon Polly, provide metrics for successful and unsuccessful requests.
- AWS IoT provides metrics for number of rule executions as well as specific success and failure metrics around the rules.
- Amazon API Gateway provides metrics for number of requests, erroneous requests, and latency for your APIs.
- AWS Personal Health Dashboard gives you a personalized view into the performance and availability of the AWS services underlying your AWS resources.

In addition, monitor all of your external endpoints from remote locations to ensure that they are independent of your base implementation. This active monitoring can be done with synthetic transactions (sometimes referred to as “user canaries”, but not to be confused with canary deployments) which periodically execute some number of common tasks performed by consumers of the application. Keep these short in duration and be sure not to overload your workflow during testing. Amazon CloudWatch Synthetics enables you to create canaries to monitor your endpoints and APIs. You can also combine the synthetic canary client nodes with AWS X-Ray console to pinpoint which synthetic canaries are experiencing issues with errors, faults, or throttling rates for the selected time frame.

Aggregation — Define and calculate metrics: Store log data and apply filters where necessary to calculate metrics, such as counts of a specific log event, or latency calculated from log event timestamps.

Amazon CloudWatch and Amazon S3 serve as the primary aggregation and storage layers. For some services, like AWS Auto Scaling and Elastic Load Balancing, default metrics are provided “out the box” for CPU load or average request latency across a cluster or instance. For streaming services, like VPC Flow Logs and AWS CloudTrail, event data is forwarded to CloudWatch Logs and you need to define and apply metrics filters to extract metrics from the event data. This gives you time series data, which can serve as inputs to CloudWatch alarms that you define to trigger alerts.

Real-time processing and alarming — Send notifications: Organizations that need to know receive notifications when significant events occur.

Alerts can also be sent to Amazon Simple Notification Service (Amazon SNS) topics, and then pushed to any number of subscribers. For example, Amazon SNS can forward alerts to an email alias so that technical staff can respond.

Real-time processing and alarming — Automate responses: Use automation to take action when an event is detected, for example, to replace failed components.

Alerts can trigger AWS Auto Scaling events, so that clusters react to changes in demand. Alerts can be sent to Amazon Simple Queue Service (Amazon SQS), which can serve as an integration point for third-party ticket systems. AWS Lambda can also subscribe to alerts, providing users an asynchronous serverless model that reacts to change dynamically. AWS Config continuously monitors and records your AWS resource configurations, and can trigger AWS Systems Manager Automation to remediate issues.

Amazon DevOps Guru can automatically monitor application resources for anomalous behavior and deliver targeted recommendations to speed up problem identification and remediation times.

Storage and Analytics: Collect log files and metrics histories and analyze these for broader trends and workload insights.

Amazon CloudWatch Logs Insights supports a simple yet powerful query language that you can use to analyze log data. Amazon CloudWatch Logs also supports subscriptions that allow data to flow seamlessly to Amazon S3 where you can use or Amazon Athena to query the data. It supports queries on a large array of formats. For more information, see Supported SerDes and Data Formats in the Amazon Athena User Guide. For analysis of huge log file sets, you can run an Amazon EMR cluster to run petabyte-scale analyses.

There are a number of tools provided by partners and third parties that allow for aggregation, processing, storage, and analytics. These tools include New Relic, Splunk, Loggly, Logstash, CloudHealth, and Nagios. However, outside generation of system and application logs is unique to each cloud provider, and often unique to each service.

An often-overlooked part of the monitoring process is data management. You need to determine the retention requirements for monitoring data, and then apply lifecycle policies accordingly. Amazon S3 supports lifecycle management at the S3 bucket level. This lifecycle management can be applied differently to different paths in the bucket. Toward the end of the lifecycle, you can transition data to Amazon S3 Glacier for long-term storage, and then expiration after the end of the retention period is reached. The S3 Intelligent-Tiering storage class is designed to optimize costs by automatically moving data to the most cost-effective access tier, without performance impact or operational overhead.

Conduct reviews regularly: Frequently review how workload monitoring is implemented and update it based on significant events and changes.

Effective monitoring is driven by key business metrics. Ensure these metrics are accommodated in your workload as business priorities change.

Auditing your monitoring helps ensure that you know when an application is meeting its availability goals. Root Cause Analysis requires the ability to discover what happened when failures occur. AWS provides services that allow you to track the state of your services during an incident:

- **Amazon CloudWatch Logs:** You can store your logs in this service and inspect their contents.
- **Amazon CloudWatch Logs Insights**: Is a fully managed service that enables you to run analyze massive logs in seconds. It gives you fast, interactive queries and visualizations.
- **AWS Config:** You can see what AWS infrastructure was in use at different points in time.
- **AWS CloudTrail:** You can see which AWS APIs were invoked at what time and by what principal.

At AWS, we conduct a weekly meeting to review operational performance and to share learnings between teams. Because there are so many teams in AWS, we created The Wheel to randomly pick a

workload to review. Establishing a regular cadence for operational performance reviews and knowledge sharing enhances your ability to achieve higher performance from your operational teams.

Monitor end-to-end tracing of requests through your system: Use AWS X-Ray or third-party tools so that developers can more easily analyze and debug distributed systems to understand how their applications and its underlying services are performing.

Resources

Documentation

- Using Amazon CloudWatch Metrics
- Using Canaries (Amazon CloudWatch Synthetics)
- Amazon CloudWatch Logs Insights Sample Queries
- AWS Systems Manager Automation
- What is AWS X-Ray?
- Debugging with Amazon CloudWatch Synthetics and AWS X-Ray
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility
- What is DevOps Guru?

Labs

- One Observability Workshop: This workshop provides hands-on experience on the wide variety of toolsets that AWS uses to set up monitoring and observability of your applications.

Design your Workload to Adapt to Changes in Demand

A scalable **workload** provides elasticity to add or remove resources automatically so that they closely match the current demand at any given point in time.

Use automation when obtaining or scaling resources: When replacing impaired resources or scaling your workload, automate the process by using managed AWS services, such as Amazon S3 and AWS Auto Scaling. You can also use third-party tools and AWS SDKs to automate scaling.

Managed AWS services include Amazon S3, Amazon CloudFront, AWS Auto Scaling, AWS Lambda, Amazon DynamoDB, AWS Fargate, and Amazon Route 53.

AWS Auto Scaling lets you detect and replace impaired instances. It also lets you build scaling plans for resources including Amazon EC2 instances and Spot Fleets, Amazon ECS tasks, Amazon DynamoDB tables and indexes, and Amazon Aurora Replicas.

When scaling EC2 instances or Amazon ECS containers hosted on EC2 instances, ensure that you use multiple Availability Zones (preferably at least three) and add or remove capacity to maintain balance across these Availability Zones.

When using AWS Lambda, they scale automatically. Every time an event notification is received for your function, AWS Lambda quickly locates free capacity within its compute fleet and runs your code up to the allocated concurrency. You need to ensure that the necessary concurrency is configured on the specific Lambda, and in your Service Quotas.

Amazon S3 automatically scales to handle high request rates. For example, your application can achieve at least 3,500 PUT/COPY/POST/DELETE or 5,500 GET/HEAD requests per second per prefix in a bucket. There are no limits to the number of prefixes in a bucket. You can increase your read or write performance by parallelizing reads. For example, if you create 10 prefixes in an Amazon S3 bucket to parallelize reads, you could scale your read performance to 55,000 read requests per second.

Configure and use Amazon CloudFront or a trusted content delivery network (CDN). A CDN can provide faster end-user response times and can serve requests for content that may cause unnecessary scaling of your workloads.

Obtain resources upon detection of impairment to a workload: Scale resources reactively when necessary if availability is impacted, so as to restore workload availability.

You first must configure health checks and the criteria on these checks to indicate when availability is impacted by lack of resources. Then either notify the appropriate personnel to manually scale the resource, or trigger automation to automatically scale it.

Scale can be manually adjusted for your workload, for example, changing the number of EC2 instances in an Auto Scaling group or modifying throughput of a DynamoDB table can be done through the console or AWS CLI. However automation should be used whenever possible (refer to **Use automation when obtaining or scaling resources**).

Obtain resources upon detection that more resources are needed for a workload: Scale resources proactively to meet demand and avoid availability impact.

Many AWS services automatically scale to meet demand. If using Amazon EC2 instances or Amazon ECS clusters, you can configure automatic scaling of these to occur based on usage metrics that correspond to demand for your workload. For Amazon EC2, average CPU utilization, load balancer request count, or network bandwidth can be used to scale out (or scale in) EC2 instances. For Amazon ECS, average CPU utilization, load balancer request count, and memory utilization can be used to scale our (or scale in) ECS tasks. Using Target Auto Scaling on AWS, the autoscaler acts like a household thermostat, adding or removing resources to maintain the target value (for example, 70% CPU utilization) that you specify.

AWS Auto Scaling can also do Predictive Auto Scaling, which uses machine learning to analyze each resource's historical workload and regularly forecasts the future load for the next two days.

Little's Law helps calculate how many instances of compute (EC2 instances, concurrent Lambda functions, etc.) that you need.

$L = \lambda W$

L = number of instances (or mean concurrency in the system)

λ = mean rate at which requests arrive (req/sec)

W = mean time that each request spends in the system (sec)

For example, at 100 rps, if each request takes 0.5 seconds to process, you will need 50 instances to keep up with demand.

Load test your workload: Adopt a load testing methodology to measure if scaling activity will meet workload requirements.

It's important to perform sustained load testing. Load tests should discover the breaking point and test performance of your workload. AWS makes it easy to set up temporary testing environments that model the scale of your production workload. In the cloud, you can create a production-scale test environment on demand, complete your testing, and then decommission the resources. Because you only pay for the test environment when it's running, you can simulate your live environment for a fraction of the cost of testing on premises.

Load testing in production should also be considered as part of game days where the production system is stressed, during hours of lower customer usage, with all personnel on hand to interpret results and address any problems that arise.

Resources

Documentation

- AWS Auto Scaling: How Scaling Plans Work
- What Is Amazon EC2 Auto Scaling?
- Managing Throughput Capacity Automatically with DynamoDB Auto Scaling
- What is Amazon CloudFront?
- Distributed Load Testing on AWS: simulate thousands of connected users
- AWS Marketplace: products that can be used with auto scaling
- APN Partner: partners that can help you create automated compute solutions

External Links

- Telling Stories About Little's Law

Implement Change

Controlled changes are necessary to deploy new functionality and to ensure that the workloads and the operating environment are running known, properly patched software. If these changes are uncontrolled, then it makes it difficult to predict the effect of these changes, or to address issues that arise because of them.

Use runbooks for standard activities such as deployment: Runbooks are the predefined steps to achieve specific outcomes. Use runbooks to perform standard activities, whether done manually or automatically. Examples include deploying a workload, patching it, or making DNS modifications.

For example, put processes in place to ensure rollback safety during deployments. Ensuring that you can roll back a deployment without any disruption for your customers is critical in making a service reliable.

For runbook procedures, start with a valid effective manual process, implement it in code, and trigger automated execution where appropriate.

Even for sophisticated workloads that are highly automated, runbooks are still useful for running game days (p. 45) or meeting rigorous reporting and auditing requirements.

Note that playbooks are used in response to specific incidents, and runbooks are used to achieve specific outcomes. Often, runbooks are for routine activities, while playbooks are used for responding to non-routine events.

Integrate functional testing as part of your deployment: Functional tests are run as part of automated deployment. If success criteria are not met, the pipeline is halted or rolled back.

These tests are run in a pre-production environment, which is staged prior to production in the pipeline. Ideally, this is done as part of a deployment pipeline.

Integrate resiliency testing as part of your deployment: Resiliency tests (as part of chaos engineering) are run as part of the automated deployment pipeline in a pre-production environment.

These tests are staged and run in the pipeline prior to production. They should also be run in production, but as part of Game Days (p. 45).

Deploy using immutable infrastructure: This is a model that mandates that no updates, security patches, or configuration changes happen *in-place* on production systems. When a change is needed, the architecture is built onto new infrastructure and deployed into production.

The most common implementation of the immutable infrastructure paradigm is the ***immutable server***. This means that if a server needs an update or a fix, new servers are deployed instead of updating the ones already in use. So, instead of logging into the server via SSH and updating the software version, every change in the application starts with a software push to the code repository, for example, git push. Since changes are not allowed in immutable infrastructure, you can be sure about the state of the deployed system. Immutable infrastructures are inherently more consistent, reliable, and predictable, and they simplify many aspects of software development and operations.

Use a canary or blue/green deployment when deploying applications in immutable infrastructures.

Canary deployment is the practice of directing a small number of your customers to the new version, usually running on a single service instance (the canary). You then deeply scrutinize any behavior changes or errors that are generated. You can remove traffic from the canary if you encounter critical problems and send the users back to the previous version. If the deployment is successful, you can continue to deploy at your desired velocity, while monitoring the changes for errors, until you are fully deployed. AWS CodeDeploy can be configured with a deployment configuration that will enable a canary deployment.

Blue/green deployment is similar to the canary deployment except that a full fleet of the application is deployed in parallel. You alternate your deployments across the two stacks (blue and green). Once again, you can send traffic to the new version, and fall back to the old version if you see problems with the deployment. Commonly all traffic is switched at once, however you can also use fractions of your traffic to each version to dial up the adoption of the new version using the weighted DNS routing capabilities of Amazon Route 53. AWS CodeDeploy and AWS Elastic Beanstalk can be configured with a deployment configuration that will enable a blue/green deployment.

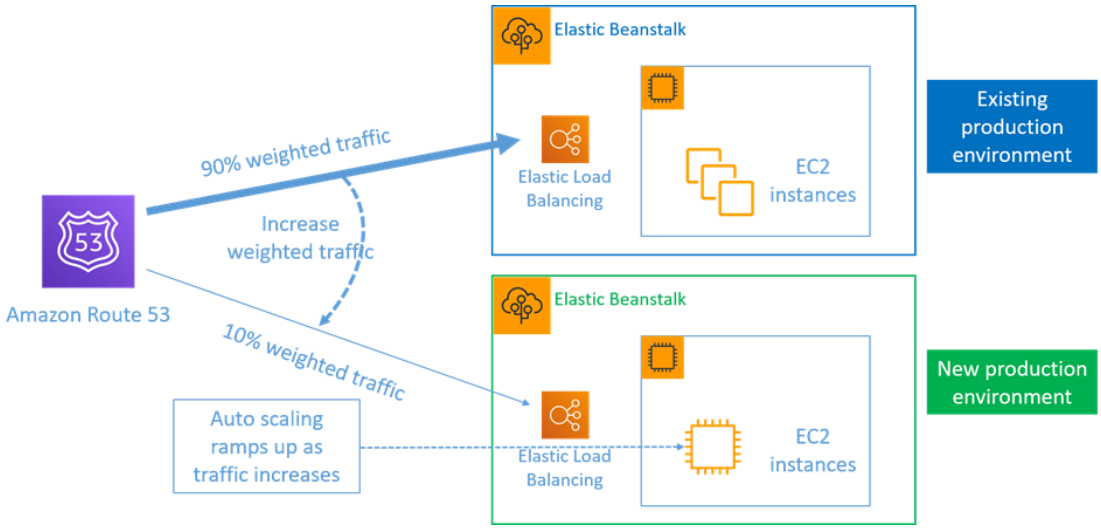


Figure 8: Blue/green deployment with AWS Elastic Beanstalk and Amazon Route 53

Benefits of immutable infrastructure:

- **Reduction in configuration drifts:** By frequently replacing servers from a base, known and version-controlled configuration, the infrastructure is **reset** to a known state, avoiding configuration drifts.
- **Simplified deployments**: Deployments are simplified because they don't need to support upgrades. Upgrades are just new deployments.

- **Reliable atomic deployments:** Deployments either complete successfully, or nothing changes. It gives more trust in the deployment process.
- **Safer deployments with fast rollback and recovery processes:** Deployments are safer because the previous working version is not changed. You can roll back to it if errors are detected.
- **Consistent testing and debugging environments:** Since all servers use the same image, there are no differences between environments. One build is deployed to multiple environments. It also prevents inconsistent environments and simplifies testing and debugging.
- **Increased scalability:** Since servers use a base image, are consistent, and repeatable, automatic scaling is trivial.
- **Simplified toolchain**: The toolchain is simplified since you can get rid of configuration management tools managing production software upgrades. No extra tools or agents are installed on servers. Changes are made to the base image, tested, and rolled-out.
- **Increased security:** By denying all changes to servers, you can disable SSH on instances and remove keys. This reduces the attack vector, improving your organization's security posture.

Deploy changes with automation: Deployments and patching are automated to eliminate negative impact.

Making changes to production systems is one of the largest risk areas for many organizations. We consider deployments a first-class problem to be solved alongside the business problems that the software addresses. Today, this means the use of automation wherever practical in operations, including testing and deploying changes, adding or removing capacity, and migrating data. AWS CodePipeline lets you manage the steps required to release your workload. This includes a deployment state using AWS CodeDeploy to automate deployment of application code to Amazon EC2 instances, on-premises instances, serverless Lambda functions, or Amazon ECS services.

Recommendation
Although conventional wisdom suggests that you keep humans in the loop for the most difficult operational procedures, we suggest that you automate the most difficult procedures for that very reason.

Additional deployment patterns to minimize risk:

Feature flags (also known as feature toggles) are configuration options on an application. You can deploy the software with a feature turned off, so that your customers don't see the feature. You can then turn on the feature, as you'd do for a canary deployment, or you can set the change pace to 100% to see the effect. If the deployment has problems, you can simply turn the feature back off without rolling back.

Fault isolated zonal deployment: One of the most important rules AWS has established for its own deployments is to avoid touching multiple Availability Zones within a Region at the same time. This is critical to ensuring that Availability Zones are independent for purposes of our availability calculations. We recommend that you use similar considerations in your deployments.

Operational Readiness Reviews (ORRs)

AWS finds it useful to perform operational readiness reviews that evaluate the completeness of the testing, ability to monitor, and importantly, the ability to audit the application's performance to its SLAs and provide data in the event of an interruption or other operational anomaly. A formal ORR is conducted prior to initial production deployment. AWS will repeat ORRs periodically (once per year, or before critical performance periods) to ensure that there has not been "drift" from operational expectations. For more information on operational readiness, see the Operational Excellence pillar of the AWS Well-Architected Framework.

Recommendation
Conduct an Operational Readiness Review (ORR) for applications prior to initial production use, and periodically thereafter.

Resources

Videos

- AWS Summit 2019: CI/CD on AWS

Documentation

- What Is AWS CodePipeline?
- What Is CodeDeploy?
- Overview of a Blue/Green Deployment
- Deploying Serverless Applications Gradually
- The Amazon Builders' Library: Ensuring rollback safety during deployments
- The Amazon Builders' Library: Going faster with continuous delivery
- AWS Marketplace: products that can be used to automate your deployments
- APN Partner: partners that can help you create automated deployment solutions

Labs

- Well-Architected lab: Level 300: Testing for Resiliency of EC2 RDS and S3

External Links

- CanaryRelease

Failure Management

> Failures are a given and everything will eventually fail over time: from routers to hard disks, from operating systems to memory units corrupting TCP packets, from transient errors to permanent failures. This is a given, whether you are using the highest-quality hardware or lowest cost components - **Werner Vogels, CTO - Amazon.com**

Low-level hardware component failures are something to be dealt with every day in in an on-premises data center. In the cloud, however, you should be protected against most of these types of failures. For example, Amazon EBS volumes are placed in a specific Availability Zone where they are automatically replicated to protect you from the failure of a single component. All EBS volumes are designed for 99.999% availability. Amazon S3 objects are stored across a minimum of three Availability Zones providing 99.999999999% durability of objects over a given year. Regardless of your cloud provider, there is the potential for failures to impact your workload. Therefore, you must take steps to implement resiliency if you need your workload to be reliable.

A prerequisite to applying the best practices discussed here is that you must ensure that the people designing, implementing, and operating your workloads are aware of business objectives and the reliability goals to achieve these. These people must be aware of and trained for these reliability requirements.

The following sections explain the best practices for managing failures to prevent impact on your workload.

Topics

- Back up Data (p. 33)
- Use Fault Isolation to Protect Your Workload (p. 35)
- Design your Workload to Withstand Component Failures (p. 40)
- Test Reliability (p. 43)
- Plan for Disaster Recovery (DR) (p. 46)

Back up Data

Back up data, applications, and configuration to meet requirements for recovery time objectives (RTO) and recovery point objectives (RPO).

Identify and back up all data that needs to be backed up, or reproduce the data from sources: Amazon S3 can be used as a backup destination for multiple data sources. AWS services like Amazon EBS, Amazon RDS, and Amazon DynamoDB have built in capabilities to create backups. Or third-party backup software can be used. Alternatively, if the data can be reproduced from other sources to meet RPO, you may not require a backup.

On-premises data can be backed up to the AWS Cloud using Amazon S3 buckets and AWS Storage Gateway. Backup data can be archived using Amazon S3 Glacier or S3 Glacier Deep Archive for affordable, non-time sensitive cloud storage.

If you have loaded data from Amazon S3 to a data warehouse (like Amazon Redshift), or MapReduce cluster (like Amazon EMR) to do analysis on that data, this may be an example of data that can be

reproduced from other sources. As long as the results of these analyses are either stored somewhere or reproducible, you would not suffer a data loss from a failure in the data warehouse or MapReduce cluster. Other examples that can be reproduced from sources include caches (like Amazon ElastiCache) or RDS read replicas.

Secure and encrypt backup: Detect access using authentication and authorization like AWS Identity and Access Management (IAM), and detect data integrity compromise by using encryption.

Amazon S3 supports several methods of encryption of your data at rest. Using server-side encryption, Amazon S3 accepts your objects as unencrypted data, and then encrypts them before persisting them. Using client-side encryption your workload application is responsible for encrypting the data before it is sent to S3. Both methods allow you to either use AWS Key Management Service (AWS KMS) to create and store the data key, or you may provide your own key (which you are then responsible for). Using AWS KMS, you can set policies using IAM on who can and cannot access your data keys and decrypted data.

For Amazon RDS, if you have chosen to encrypt your databases, then your backups are encrypted also. DynamoDB backups are always encrypted.

Perform data backup automatically: Configure backups to be made automatically based on a periodic schedule, or by changes in the dataset. RDS instances, EBS volumes, DynamoDB tables, and S3 objects can all be configured for automatic backup. AWS Marketplace solutions or third-party solutions can also be used.

Amazon Data Lifecycle Manager can be used to automate EBS snapshots. Amazon RDS and Amazon DynamoDB enable continuous backup with Point in Time Recovery. Amazon S3 versioning, once enabled, is automatic.

For a centralized view of your backup automation and history, AWS Backup provides a fully managed, policy-based backup solution. It centralizes and automates the back up of data across multiple AWS services in the cloud as well as on premises using the AWS Storage Gateway.

In additional to versioning, Amazon S3 features replication. The entire S3 bucket can be automatically replicated to another bucket in a different AWS Region.

Perform periodic recovery of the data to verify backup integrity and processes: Validate that your backup process implementation meets your recovery time objective (RTO) and recovery point objective (RPO) by performing a recovery test.

Using AWS, you can stand up a testing environment and restore your backups there to assess RTO and RPO capabilities, and run tests on data content and integrity.

Additionally, Amazon RDS and Amazon DynamoDB allow point-in-time recovery (PITR). Using continuous backup, you are able to restore your dataset to the state it was in at a specified date and time.

Resources

Videos

- AWS re:Invent 2019: Deep dive on AWS Backup, ft. Rackspace (STG341)

Documentation

- What Is AWS Backup?
- Amazon S3: Protecting Data Using Encryption
- Encryption for Backups in AWS

- On-demand backup and restore for DynamoDB
- EFS-to-EFS backup
- AWS Marketplace: products that can be used for backup
- APN Partner: partners that can help with backup

Labs

- Well-Architected lab: Level 200: Testing Backup and Restore of Data
- Well-Architected lab: Level 200: Implementing Bi-Directional Cross-Region Replication (CRR) for Amazon Simple Storage Service (Amazon S3)

Use Fault Isolation to Protect Your Workload

Fault isolated boundaries limit the effect of a failure within a workload to a limited number of components. Components outside of the boundary are unaffected by the failure. Using multiple fault isolated boundaries, you can limit the impact on your workload.

Deploy the workload to multiple locations: Distribute workload data and resources across multiple Availability Zones or, where necessary, across AWS Regions. These locations can be as diverse as required.

One of the bedrock principles for service design in AWS is the avoidance of single points of failure in underlying physical infrastructure. This motivates us to build software and systems that use multiple Availability Zones and are resilient to failure of a single zone. Similarly, systems are built to be resilient to failure of a single compute node, single storage volume, or single instance of a database. When building a system that relies on redundant components, it's important to ensure that the components operate independently, and in the case of AWS Regions, autonomously. The benefits achieved from theoretical availability calculations with redundant components are only valid if this holds true.

Availability Zones (AZs)

AWS Regions are composed of multiple Availability Zones that are designed to be independent of each other. Each Availability Zone is separated by a meaningful physical distance from other zones to avoid correlated failure scenarios due to environmental hazards like fires, floods, and tornadoes. Each Availability Zone also has independent physical infrastructure: dedicated connections to utility power, standalone backup power sources, independent mechanical services, and independent network connectivity within and beyond the Availability Zone. This design limits faults in any of these systems to just the one affected AZ. Despite being geographically separated, Availability Zones are located in the same regional area which enables high-throughput, low-latency networking. The entire AWS region (across all Availability Zones, consisting of multiple physically independent data centers) can be treated as a single logical deployment target for your workload, including the ability to synchronously replicate data (for example, between databases). This allows you to use Availability Zones in an active/active or active/standby configuration.

Availability Zones are independent, and therefore workload availability is increased when the workload is architected to use multiple zones. Some AWS services (including the Amazon EC2 instance data plane) are deployed as strictly zonal services where they have shared fate with the Availability Zone they are in. Amazon EC2 instance in the other AZs will however be unaffected and continue to function. Similarly, if a failure in an Availability Zone causes an Amazon Aurora database to fail, a read-replica Aurora instance in an unaffected AZ can be automatically promoted to primary. Regional AWS services, like Amazon DynamoDB on the other hand internally use multiple Availability Zones in an active/active configuration to achieve the availability design goals for that service, without you needing to configure AZ placement.

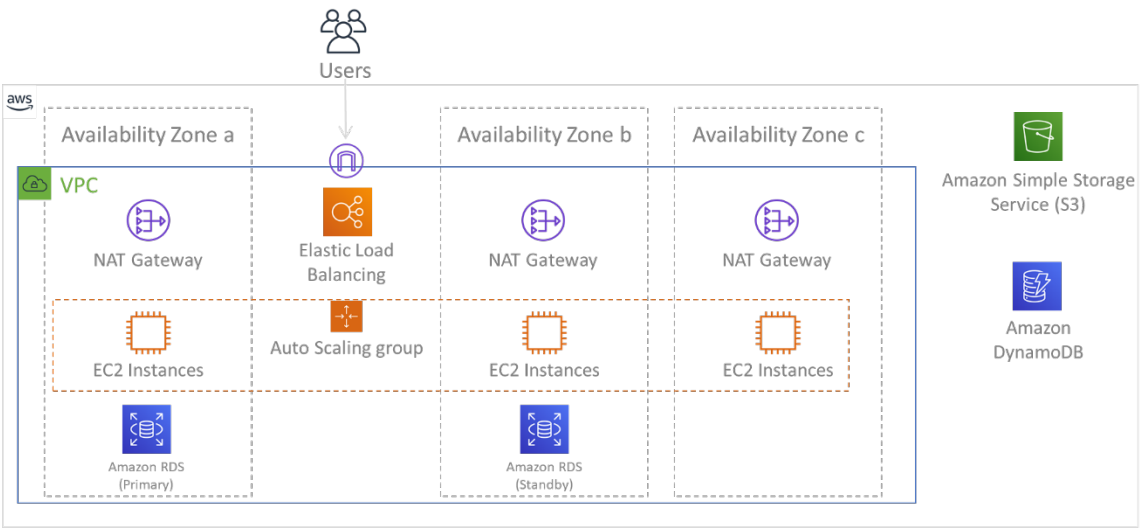


Figure 9: Multi-tier architecture deployed across three Availability Zones. Note that Amazon S3 and Amazon DynamoDB are always Multi-AZ automatically. The ELB also is deployed to all three zones.

While AWS control planes typically provide the ability to manage resources within the entire Region (multiple Availability Zones), certain control planes (including Amazon EC2 and Amazon EBS) have the ability to filter results to a single Availability Zone. When this is done, the request is processed only in the specified Availability Zone, reducing exposure to disruption in other Availability Zones. In this AWS CLI example, it illustrates getting Amazon EC2 instance information from only the `us-east-2c` Availability Zone:

```
AWS ec2 describe-instances --filters Name=availability-zone,Values=us-east-2c
```

AWS Local Zones

AWS Local Zones act similarly to Availability Zones within their respective AWS Region in that they can be selected as a placement location for zonal AWS resources like subnets and EC2 instances. What makes them special is that they are located not in the associated AWS Region, but near large population, industry, and IT centers where no AWS Region exists today. Yet they still retain high-bandwidth, secure connection between local workloads in the local zone and those running in the AWS Region. You should use AWS Local Zones to deploy workloads closer to your users for low-latency requirements.

Amazon Global Edge Network

Amazon Global Edge Network consists of edge locations in cities around the world. Amazon CloudFront uses this network to deliver content to end users with lower latency. AWS Global Accelerator enables you to create your workload endpoints in these edge locations to provide onboarding to the AWS global network close to your users. Amazon API Gateway enables edge-optimized API endpoints using a CloudFront distribution to facilitate client access through the closest edge location.

AWS Regions

AWS Regions are designed to be autonomous, therefore, to use a multi-region approach you would deploy dedicated copies of services to each Region.

A multi-region approach is common for *disaster recovery* strategies to meet recovery objectives when one-off large-scale events occur. See Plan for Disaster Recovery (DR) (p. 46) for more information on these strategies. Here however, we focus instead on *availability*, which seeks to deliver a mean

uptime objective over time. For high-availability objectives, a multi-region architecture will generally be designed to be active/active, where each service copy (in their respective regions) is active (serving requests).

Recommendation
Availability goals for most workloads can be satisfied using a Multi-AZ strategy within a single AWS Region. Consider multi-region architectures only when workloads have extreme availability requirements, or other business goals, that require a multi-region architecture.

AWS provides customers the capabilities to operate services cross-region. For example AWS provides continuous, asynchronous data replication of data using Amazon Simple Storage Service (Amazon S3) Replication, Amazon RDS Read Replicas (including Aurora Read Replicas), and Amazon DynamoDB Global Tables. With continuous replication, versions of your data are available for immediate use in each of your active Regions.

Using AWS CloudFormation, you can define your infrastructure and deploy it consistently across AWS accounts and across AWS Regions. And AWS CloudFormation StackSets extends this functionality by enabling you to create, update, or delete AWS CloudFormation stacks across multiple accounts and regions with a single operation. For Amazon EC2 instance deployments, an AMI (Amazon Machine Image) is used to supply information such as hardware configuration and installed software. You can implement an Amazon EC2 Image Builder pipeline that creates the AMIs you need and copy these to your active regions. This ensures that these "Golden AMIs" have everything you need to deploy and scale-out your workload in each new region.

To route traffic, both Amazon Route 53 and AWS Global Accelerator enable the definition of policies that determine which users go to which active regional endpoint. With Global Accelerator you set a traffic dial to control the percentage of traffic that is directed to each application endpoint. Route 53 supports this percentage approach, and also multiple other available policies including geoproximity and latency based ones. Global Accelerator automatically leverages the extensive network of AWS edge servers, to onboard traffic to the AWS network backbone as soon as possible, resulting in lower request latencies.

All of these capabilities operate so as to preserve each Region's autonomy. There are very few exceptions to this approach, including our services that provide global edge delivery (such as Amazon CloudFront and Amazon Route 53), along with the control plane for the AWS Identity and Access Management (IAM) service. The vast majority of services operate entirely within a single Region.

On-premises data center

For workloads that run in an on-premises data center, architect a hybrid experience when possible. AWS Direct Connect provides a dedicated network connection from your premises to AWS enabling you to run in both.

Another option is to run AWS infrastructure and services on premises using AWS Outposts. AWS Outposts is a fully managed service that extends AWS infrastructure, AWS services, APIs, and tools to your data center. The same hardware infrastructure used in the AWS Cloud is installed in your data center. AWS Outposts are then connected to the nearest AWS Region. You can then use AWS Outposts to support your workloads that have low latency or local data processing requirements.

Automate recovery for components constrained to a single location: If components of the workload can only run in a single Availability Zone or on-premises data center, you must implement the capability to do a complete rebuild of the workload within defined recovery objectives.

If the best practice to deploy the workload to multiple locations is not possible due to technological constraints, you must implement an alternate path to resiliency. You must automate the ability to recreate necessary infrastructure, redeploy applications, and recreate necessary data for these cases.

For example, Amazon EMR launches all nodes for a given cluster in the same Availability Zone because running a cluster in the same zone improves performance of the jobs flows as it provides a higher data access rate. If this component is required for workload resilience, then you must have a way to re-deploy the cluster and its data. Also for Amazon EMR, you should provision redundancy in ways other than using Multi-AZ. You can provision multiple master nodes. Using EMR File System (EMRFS), data in EMR can be stored in Amazon S3, which in turn can be replicated across multiple Availability Zones or AWS Regions.

Similarly for Amazon Redshift, by default it provisions your cluster in a randomly selected Availability Zone within the AWS Region that you select. All the cluster nodes are provisioned in the same zone.

Use bulkhead architectures to limit scope of impact: Like the bulkheads on a ship, this pattern ensures that a failure is contained to a small subset of requests/users so that the number of impaired requests is limited, and most can continue without error. Bulkheads for data are often called *partitions*, while bulkheads for services are known as *cells*.

In a *cell-based architecture*, each cell is a complete, independent instance of the service and has a fixed maximum size. As load increases, workloads grow by adding more cells. A partition key is used on incoming traffic to determine which cell will process the request. Any failure is contained to the single cell it occurs in, so that the number of impaired requests is limited as other cells continue without error. It is important to identify the proper partition key to minimize cross-cell interactions and avoid the need to involve complex mapping services in each request. Services that require complex mapping end up merely shifting the problem to the mapping services, while services that require cross-cell interactions create dependencies between cells (and thus reduce the assumed availability improvements of doing so).

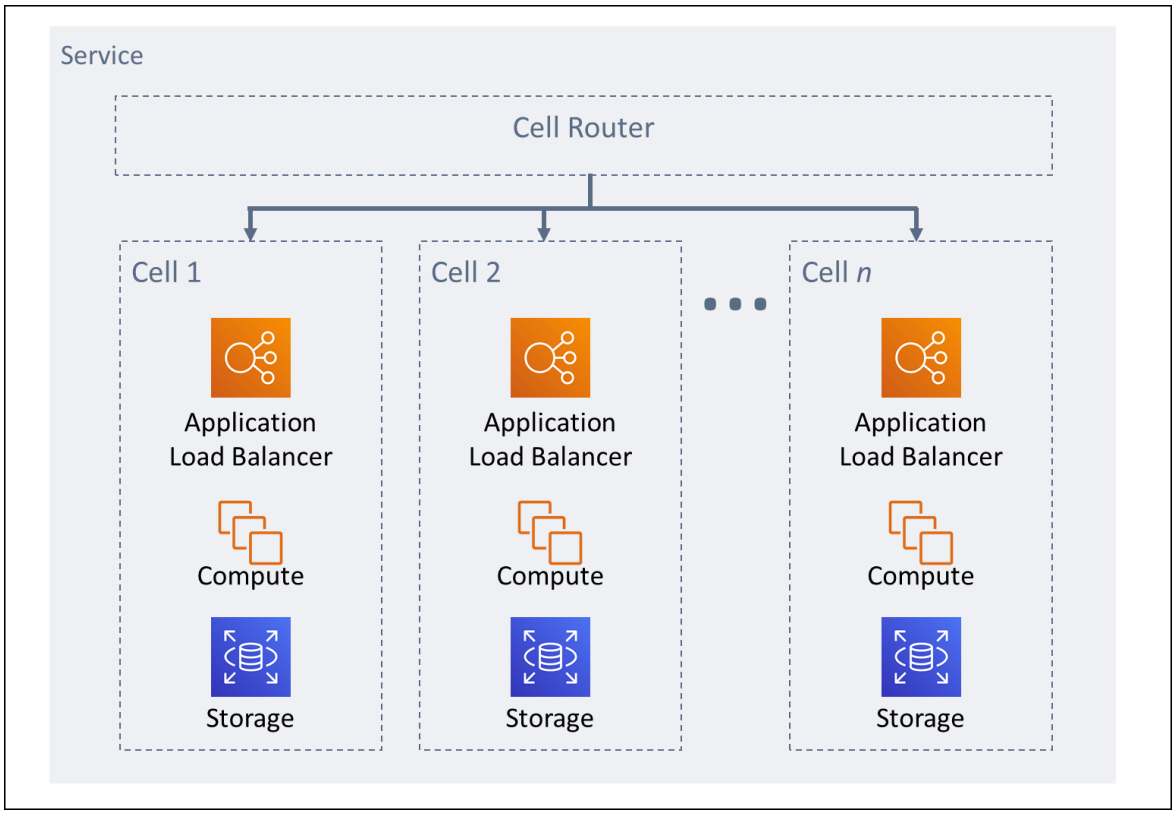


Figure 10: Cell-based architecture

In his AWS blog post, Colm MacCarthaigh explains how Amazon Route 53 uses the concept of shuffle sharding to isolate customer requests into shards. A shard in this case consists of two or more cells. Based on partition key, traffic from a customer (or resources, or whatever you want to isolate) is routed to its assigned shard. In the case of eight cells with two cells per shard, and customers divided among the four shards, 25% of customers would experience impact in the event of a problem.

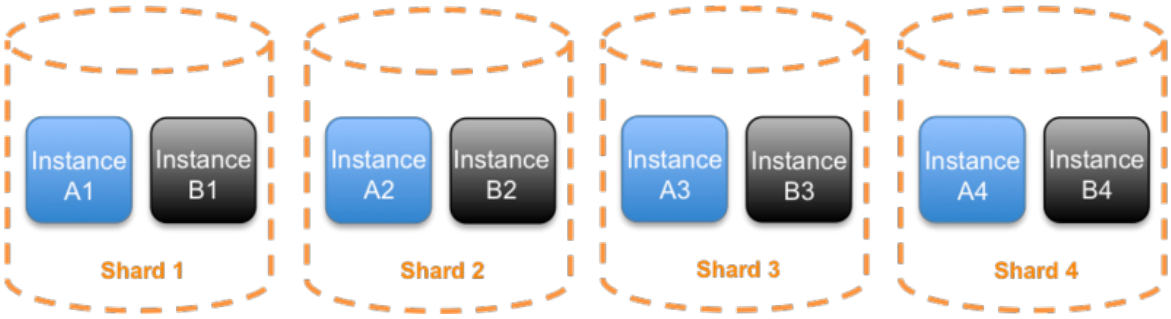


Figure 11: Service divided into four traditional shards of two cells each

With shuffle sharding, you create virtual shards of two cells each, and assign your customers to one of those virtual shards. When a problem happens, you can still lose a quarter of the whole service, but the way that customers or resources are assigned means that the scope of impact with shuffle sharding is considerably smaller than 25%. With eight cells, there are 28 unique combinations of two cells, which means that there are 28 possible shuffle shards (virtual shards). If you have hundreds or thousands of customers, and assign each customer to a shuffle shard, then the scope of impact due to a problem is just 1/28th. That's seven times better than regular sharding.

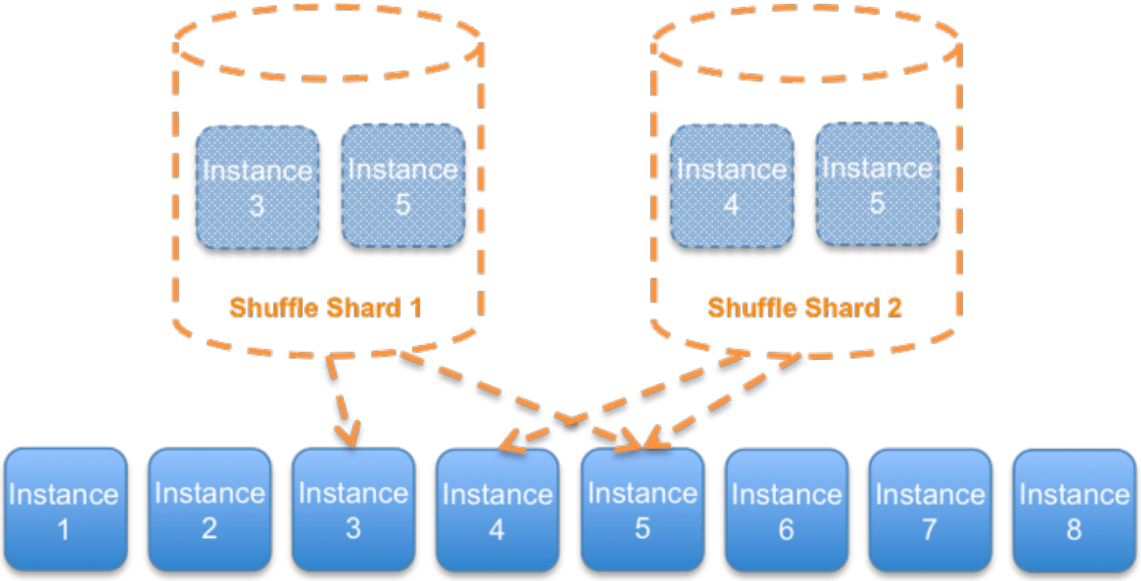


Figure 12: Service divided into 28 shuffle shards (virtual shards) of two cells each (only two shuffle shards out of the 28 possible are shown)

A shard can be used for servers, queues, or other resources in addition to cells.

Resources

Videos

- AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)
- Shuffle-sharding: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)
- AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338)
- AWS re:Invent 2019: Innovation and operation of the AWS global network infrastructure (NET339)

Documentation

- What is AWS Outposts?
- Global Tables: Multi-Region Replication with DynamoDB
- AWS Local Zones FAQ
- AWS Global Infrastructure
- Regions, Availability Zones, and Local Zones
- The Amazon Builders' Library: Workload isolation using shuffle-sharding

Design your Workload to Withstand Component Failures

Workloads with a requirement for high availability and low mean time to recovery (MTTR) must be architected for resiliency.

Monitor all components of the workload to detect failures: Continuously monitor the health of your workload so that you and your automated systems are aware of degradation or complete failure as soon as they occur. Monitor for key performance indicators (KPIs) based on business value.

All recovery and healing mechanisms must start with the ability to detect problems quickly. Technical failures should be detected first so that they can be resolved. However, availability is based on the ability of your workload to deliver business value, so this needs to be a key measure of your detection and remediation strategy.

Failover to healthy resources: Ensure that if a resource failure occurs, that healthy resources can continue to serve requests. For location failures (such as Availability Zone or AWS Region) ensure you have systems in place to failover to healthy resources in unimpaired locations.

This is easier for individual resource failures (such as an EC2 instance) or impairment of an Availability Zone in a multi-AZ workload, as AWS services, such as Elastic Load Balancing and AWS Auto Scaling, help distribute load across resources and Availability Zones. For multi-region workloads, this is more complicated. For example, cross-region read replicas enable you to deploy your data to multiple AWS Regions, but you still must promote the read replica to primary and point your traffic at it in the event of a primary location failure. Amazon Route 53 and AWS Global Accelerator can also help route traffic across AWS Regions.

If your workload is using AWS services, such as Amazon S3 or Amazon DynamoDB, then they are automatically deployed to multiple Availability Zones. In case of failure, the AWS control plane automatically routes traffic to healthy locations for you. For Amazon RDS, you must choose Multi-AZ as a configuration option, and then on failure AWS automatically directs traffic to the healthy instance. For Amazon EC2 instances or Amazon ECS tasks, you choose which Availability Zones to deploy to. Elastic Load Balancing then provides the solution to detect instances in unhealthy zones and route traffic to the healthy ones. Elastic Load Balancing can even route traffic to components in your on-premises data center.

For Multi-Region approaches (which might also include on-premises data centers), Amazon Route 53 provides a way to define internet domains, and assign routing policies that can include health checks to ensure that traffic is routed to healthy regions. Alternately, AWS Global Accelerator provides static IP addresses that act as a fixed entry point to your application, then routes to endpoints in AWS Regions of your choosing, using the AWS global network instead of the internet for better performance and reliability.

AWS approaches the design of our services with fault recovery in mind. We design services to minimize the time to recover from failures and impact on data. Our services primarily use data stores that acknowledge requests only after they are durably stored across multiple replicas. These services and resources include Amazon Aurora, Amazon Relational Database Service (Amazon RDS) Multi-AZ DB instances, Amazon S3, Amazon DynamoDB, Amazon Simple Queue Service (Amazon SQS), and Amazon Elastic File System (Amazon EFS). They are constructed to use cell-based isolation and use the fault isolation provided by Availability Zones. We use automation extensively in our operational procedures. We also optimize our replace-and-restart functionality to recover quickly from interruptions.

Automate healing on all layers: Upon detection of a failure, use automated capabilities to perform actions to remediate.

Ability to restart is an important tool to remediate failures. As discussed previously for distributed systems, a best practice is to make services stateless where possible. This prevents loss of data or availability on restart. In the cloud, you can (and generally should) replace the entire resource (for example, EC2 instance, or Lambda function) as part of the restart. The restart itself is a simple and reliable way to recover from failure. Many different types of failures occur in workloads. Failures can occur in hardware, software, communications, and operations. Rather than constructing novel mechanisms to trap, identify, and correct each of the different types of failures, map many different categories of failures to the same recovery strategy. An instance might fail due to hardware failure, an operating system bug, memory leak, or other causes. Rather than building custom remediation for each situation, treat any of them as an instance failure. Terminate the instance, and allow AWS Auto Scaling to replace it. Later, carry out the analysis on the failed resource out of band.

Another example is the ability to restart a network request. Apply the same recovery approach to both a network timeout and a dependency failure where the dependency returns an error. Both events have a similar effect on the system, so rather than attempting to make either event a “special case”, apply a similar strategy of limited retry with exponential backoff and jitter.

Ability to restart is a recovery mechanism featured in Recovery Oriented Computing (ROC) and high availability cluster architectures.

Amazon EventBridge can be used to monitor and filter for events such as CloudWatch Alarms or changes in state in other AWS services. Based on event information, it can then trigger AWS Lambda (or other targets) to execute custom remediation logic on your workload.

Amazon EC2 Auto Scaling can be configured to check for EC2 instance health. If the instance is in any state other than running, or if the system status is impaired, Amazon EC2 Auto Scaling considers the instance to be unhealthy and launches a replacement instance. If using AWS OpsWorks, you can configure Auto Healing of EC2 instances at the layer level.

For large-scale replacements (such as the loss of an entire Availability Zone), static stability is preferred for high availability instead of trying to obtain multiple new resources at once.

Use static stability to prevent bimodal behavior: Bimodal behavior is when your workload exhibits different behavior under normal and failure modes, for example, relying on launching new instances if an Availability Zone fails. You should instead build systems that are statically stable and operate in only one mode. In this case, provision enough instances in each zone to handle workload load if one zone were removed and then use Elastic Load Balancing or Amazon Route 53 health checks to shift load away from the impaired instances.

Static stability for compute deployment (such as EC2 instances or containers) will result in the highest reliability. This must be weighed against cost concerns. It’s less expensive to provision less compute capacity and rely on launching new instances in the case of a failure. But for large-scale failures (such as an Availability Zone failure) this approach is less effective because it relies on reacting to impairments as they happen, rather than being prepared for those impairments before they happen. Your solution should weigh reliability versus the cost needs for your workload. By using more Availability Zones, the amount of additional compute you need for static stability decreases.

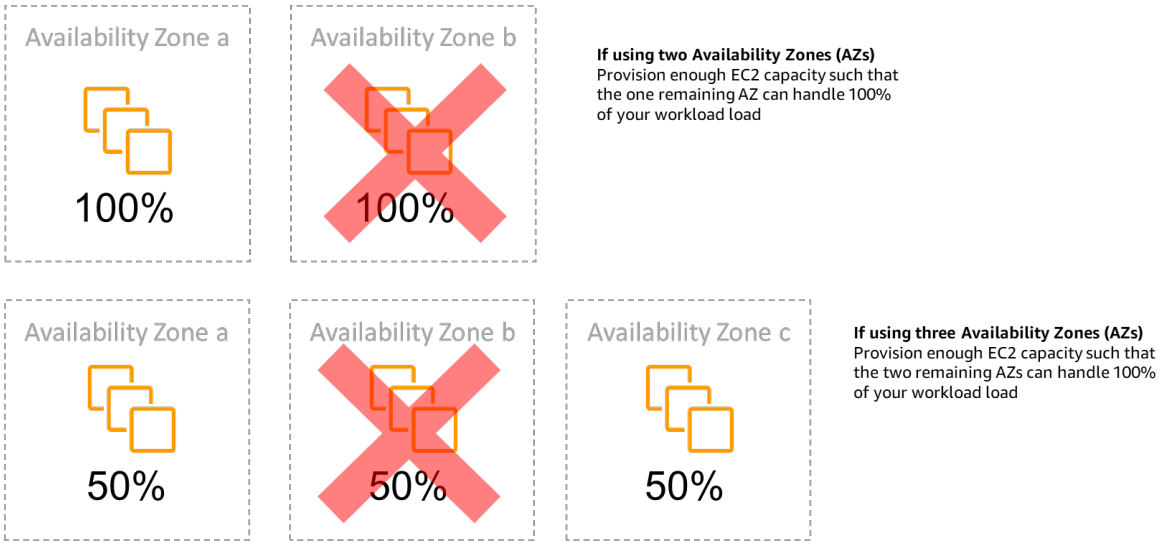


Figure 13: Static stability of EC2 instances across Availability Zones

After traffic has shifted, use AWS Auto Scaling to asynchronously replace instances from the failed zone and launch them in the healthy zones.

Another example of bimodal behavior would be a network timeout that could cause a system to attempt to refresh the configuration state of the entire system. This would add unexpected load to another component, and might cause it to fail, triggering other unexpected consequences. This negative feedback loop impacts availability of your workload. Instead, you should build systems that are statically stable and operate in only one mode. A statically stable design would be to do constant work (p. 18), and always refresh the configuration state on a fixed cadence. When a call fails, the workload uses the previously cached value, and triggers an alarm.

Another example of bimodal behavior is allowing clients to bypass your workload cache when failures occur. This might seem to be a solution that accommodates client needs, but should not be allowed because it significantly changes the demands on your workload and is likely to result in failures.

Send notifications when events impact availability: Notifications are sent upon the detection of significant events, even if the issue caused by the event was automatically resolved.

Automated healing enables your workload to be reliable. However, it can also obscure underlying problems that need to be addressed. Implement appropriate monitoring and events so that you can detect patterns of problems, including those addressed by auto healing, so that you can resolve root cause issues. Amazon CloudWatch Alarms can be triggered based on failures that occur. They can also trigger based on automated healing actions executed. CloudWatch Alarms can be configured to send emails, or to log incidents in third-party incident tracking systems using Amazon SNS integration.

Resources

Videos

- Static stability in AWS: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)

Documentation

- AWS OpsWorks: Using Auto Healing to Replace Failed Instances

- What Is Amazon EventBridge?
- Amazon Route 53: Choosing a Routing Policy
- What Is AWS Global Accelerator?
- The Amazon Builders' Library: Static stability using Availability Zones
- The Amazon Builders' Library: Implementing health checks
- AWS Marketplace: products that can be used for fault tolerance
- APN Partner: partners that can help with automation of your fault tolerance

Labs

- Well-Architected lab: Level 300: Implementing Health Checks and Managing Dependencies to Improve Reliability

External Links

- The Berkeley/Stanford Recovery-Oriented Computing (ROC) Project

Test Reliability

After you have designed your workload to be resilient to the stresses of production, testing is the only way to ensure that it will operate as designed, and deliver the resiliency you expect.

Test to validate that your workload meets functional and non-functional requirements, because bugs or performance bottlenecks can impact the reliability of your workload. Test the resiliency of your workload to help you find latent bugs that only surface in production. Exercise these tests regularly.

Use playbooks to investigate failures: Enable consistent and prompt responses to failure scenarios that are not well understood, by documenting the investigation process in playbooks. Playbooks are the predefined steps performed to identify the factors contributing to a failure scenario. The results from any process step are used to determine the next steps to take until the issue is identified or escalated.

The playbook is proactive planning that you must do, so as to be able to take reactive actions effectively. When failure scenarios not covered by the playbook are encountered in production, first address the issue (put out the fire). Then go back and look at the steps you took to address the issue and use these to add a new entry in the playbook.

Note that playbooks are used in response to specific incidents, while runbooks are used to achieve specific outcomes. Often, runbooks are used for routine activities and playbooks are used to respond to non-routine events.

Perform post-incident analysis: Review customer-impacting events, and identify the contributing factors and preventative action items. Use this information to develop mitigations to limit or prevent recurrence. Develop procedures for prompt and effective responses. Communicate contributing factors and corrective actions as appropriate, tailored to target audiences.

Assess why existing testing did not find the issue. Add tests for this case if tests do not already exist.

Test functional requirements: These include unit tests and integration tests that validate required functionality.

You achieve the best outcomes when these tests are run automatically as part of build and deployment actions. For instance, using AWS CodePipeline, developers commit changes to a source repository where

CodePipeline automatically detects the changes. Those changes are built, and tests are run. After the tests are complete, the built code is deployed to staging servers for testing. From the staging server, CodePipeline runs more tests, such as integration or load tests. Upon the successful completion of those tests, CodePipeline deploys the tested and approved code to production instances.

Additionally, experience shows that synthetic transaction testing (also known as “canary testing”, but not to be confused with canary deployments) that can run and simulate customer behavior is among the most important testing processes. Run these tests constantly against your workload endpoints from diverse remote locations. Amazon CloudWatch Synthetics enables you to create canaries to monitor your endpoints and APIs.

Test scaling and performance requirements: This includes load testing to validate that the workload meets scaling and performance requirements.

In the cloud, you can create a production-scale test environment on demand for your workload. If you run these tests on scaled down infrastructure, you must scale your observed results to what you think will happen in production. Load and performance testing can also be done in production if you are careful not to impact actual users, and tag your test data so it does not comingle with real user data and corrupt usage statistics or production reports.

With testing, ensure that your base resources, scaling settings, service quotas, and resiliency design operate as expected under load.

Test resiliency using chaos engineering: Run tests that inject failures regularly into pre-production and production environments. Hypothesize how your workload will react to the failure, then compare your hypothesis to the testing results and iterate if they do not match. Ensure that production testing does not impact users.

In the cloud, you can test how your workload fails, and you can validate your recovery procedures. Use automation to simulate different failures or to recreate scenarios that led to failures before. This exposes failure pathways that you can test and fix before a real failure scenario occurs, thus reducing risk.

> Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system’s capability to withstand turbulent conditions in production. – Principles of Chaos Engineering

In pre-production and testing environments, chaos engineering should be done regularly, and be part of your CI/CD cycle. Chaos engineering in production is encouraged, however teams must take care not to disrupt availability for customers.

Test for component failures that you have designed your workload to be resilient against. These include loss of EC2 instances, failure of the primary Amazon RDS database instance, and Availability Zone outages.

Test for external dependency unavailability. Your workload’s resiliency to transient failures of dependencies should be tested for durations that may last from less than a second to hours.

Other modes of degradation might cause reduced functionality and slow responses, often resulting in a brownout of your services. Common sources of this degradation are increased latency on critical services and unreliable network communication (dropped packets). Test for these failures, including networking effects, such as latency and dropped messages, as well as DNS failures, such as being unable to resolve a name or not being able to establish connections to dependent services.

AWS Fault Injection Simulator (AWS FIS) is a fully managed service for running fault injection experiments that can be used as part of your CD pipeline, during game days, or ad hoc. It supports gradually and simultaneously impairing performance of different types of resources including Amazon EC2, Amazon ECS, Amazon EKS, and Amazon RDS. It can also stress CPU or memory on your workload in AWS. Since it is integrated with Amazon CloudWatch Alarms, you can set guardrails to rollback a test if it causes unexpected impact.

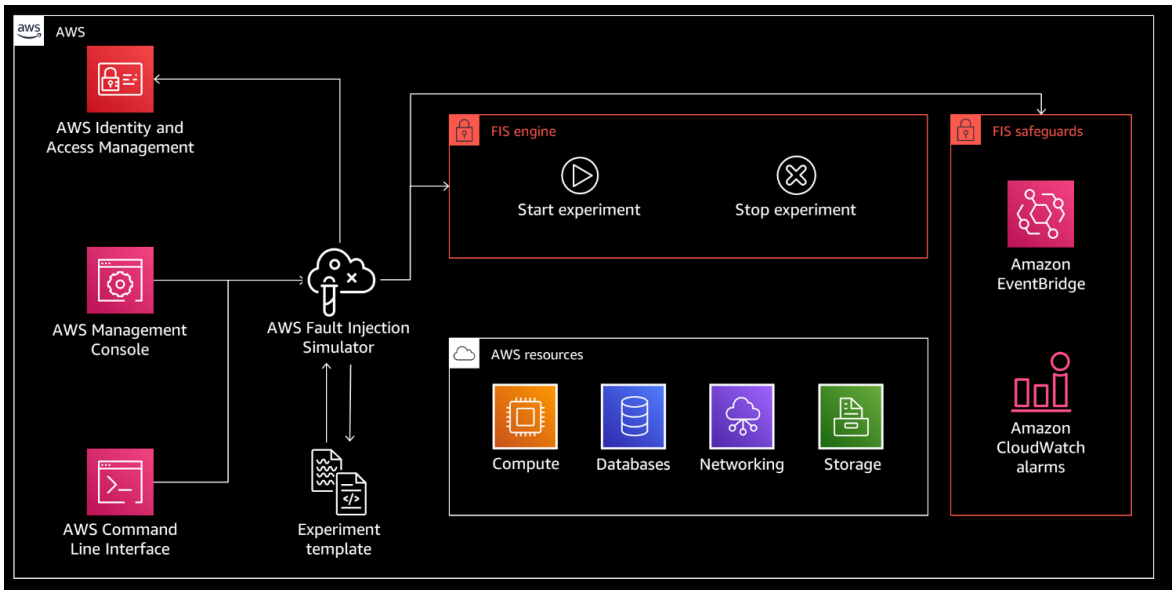


Figure 14: AWS Fault Injection Simulator (AWS FIS) integrates with AWS resources to enable you to run fault injection experiments for your workloads.

There are also several third-party options for fault injection experiments. These include open source options such as The Chaos ToolKit, and Shopify Toxiproxy, as well as commercial options like Gremlin. For organizations new to chaos engineering, it can build confidence to first use self-authored scripts. This enables engineering teams to become comfortable with how chaos is introduced into their workloads. For examples of these, see Testing for Resiliency of EC2 RDS and S3 using multiple languages such as a Bash, Python, Java, and PowerShell. You can also implement Injecting Chaos to Amazon EC2 using AWS Systems Manager, which enables you to simulate brownouts and high CPU conditions using AWS Systems Manager Documents.

Conduct game days regularly: Use game days to regularly exercise your procedures for responding to events and failures as close to production as possible (including in production environments) with the people who will be involved in actual failure scenarios. Game days enforce measures to ensure that production events do not impact users.

Game days simulate a failure or event to test systems, processes, and team responses. The purpose is to actually perform the actions the team would perform as if an exceptional event happened. This will help you understand where improvements can be made and can help develop organizational experience in dealing with events. These should be conducted regularly so that your team builds "muscle memory" on how to respond.

After your design for resiliency is in place and has been tested in non-production environments, a game day is the way to ensure that everything works as planned in production. A game day, especially the first one, is an “all hands on deck” activity where engineers and operations are all informed when it will happen, and what will occur. Runbooks are in place. Simulated events are executed, including possible failure events, in the production systems in the prescribed manner, and impact is assessed. If all systems operate as designed, detection and self-healing will occur with little to no impact. However, if negative impact is observed, the test is rolled back and the workload issues are remedied, manually if necessary (using the runbook). Since game days often take place in production, all precautions should be taken to ensure that there is no impact on availability to your customers.

Resources

Videos

- AWS re:Invent 2019: Improving resiliency with chaos engineering (DOP309-R1)

Documentation

- AWS Fault Injection Simulator (AWS FIS)
- Continuous Delivery and Continuous Integration
- Using Canaries (Amazon CloudWatch Synthetics)
- Use CodePipeline with AWS CodeBuild to test code and run builds
- Automate your operational playbooks with AWS Systems Manager
- AWS Marketplace: products that can be used for continuous integration
- APN Partner: partners that can help with implementation of a continuous integration pipeline

Labs

- Well-Architected lab: Level 300: Testing for Resiliency of EC2 RDS and S3

External Links

- Principles of Chaos Engineering
- Resilience Engineering: Learning to Embrace Failure
- Apache JMeter

Books

- Casey Rosenthal, Nora Jones. "Chaos Engineering" (April 2020)

Plan for Disaster Recovery (DR)

Having backups and redundant workload components in place is the start of your DR strategy. RTO and RPO are your objectives (p. 6) for restoration of your workload. Set these based on business needs. Implement a strategy to meet these objectives, considering locations and function of workload resources and data. The probability of disruption and cost of recovery are also key factors that help to inform the business value of providing disaster recovery for a workload.

Both Availability and Disaster Recovery rely on the same best practices such as monitoring for failures, deploying to multiple locations, and automatic failover. However Availability focuses on components of the workload, while Disaster Recovery focuses on discrete copies of the entire workload. Disaster Recovery has different objectives from Availability, focusing on time to recovery after a disaster.

Define recovery objectives for downtime and data loss: The workload has a recovery time objective (RTO) and recovery point objective (RPO).

Recovery Time Objective (RTO) is defined by the organization. RTO is the maximum acceptable delay between the interruption of service and restoration of service. This determines what is considered an acceptable time window when service is unavailable.

Recovery Point Objective (RPO) is defined by the organization. RPO is the maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

Use defined recovery strategies to meet the recovery objectives: A disaster recovery (DR) strategy has been defined to meet objectives. Choose a strategy such as: backup and restore, active/passive (pilot light or warm standby), or active/active.

When architecting a multi-region disaster recovery strategy for your workload, you should choose one of the following multi-region strategies. They are listed in increasing order of complexity, and decreasing order of RTO and RPO. *DR Region* refers to an AWS Region other than the one primary used for your workload (or any AWS Region if your workload is on premises).

> Some workloads have regulatory data residency requirements. If this applies to your workload in a locality that currently has only one AWS region, then you can use the Availability Zones within that region as discrete locations instead of AWS regions.

- **Backup and restore** (RPO in hours, RTO in 24 hours or less): Back up your data and applications using point-in-time backups into the DR Region. Restore this data when necessary to recover from a disaster.
- **Pilot light** (RPO in minutes, RTO in hours): Replicate your data from one region to another and provision a copy of your core workload infrastructure. Resources required to support data replication and backup such as databases and object storage are always on. Other elements such as application servers are loaded with application code and configurations, but are switched off and are only used during testing or when Disaster Recovery failover is invoked.
- **Warm standby** (RPO in seconds, RTO in minutes): Maintain a scaled-down but fully functional version of your workload always running in the DR Region. Business-critical systems are fully duplicated and are always on, but with a scaled down fleet. When the time comes for recovery, the system is scaled up quickly to handle the production load. The more scaled-up the Warm Standby is, the lower RTO and control plane reliance will be. When scaled up to full scale this is known as a **Hot Standby**.
- **Multi-region (multi-site) active-active** (RPO near zero, RTO potentially zero): Your workload is deployed to, and actively serving traffic from, multiple AWS Regions. This strategy requires you to synchronize data across Regions. Possible conflicts caused by writes to the same record in two different regional replicas must be avoided or handled. Data replication is useful for data synchronization and will protect you against some types of disaster, but it will not protect you against data corruption or destruction unless your solution also includes options for point-in-time recovery. Use services like Amazon Route 53 or AWS Global Accelerator to route your user traffic to where your workload is healthy. For more details on AWS services you can use for active-active architectures see the **AWS Regions** section of Use Fault Isolation to Protect Your Workload (p. 35).

Recommendation
The difference between Pilot Light and Warm Standby can sometimes be difficult to understand. Both include an environment in your DR Region with copies of your primary region assets. The distinction is that Pilot Light cannot process requests without additional action taken first, while Warm Standby can handle traffic (at reduced capacity levels) immediately. Pilot Light will require you to turn on servers, possibly deploy additional (non-core) infrastructure, and scale up, while Warm Standby only requires you to scale up (everything is already deployed and running). Choose between these based on your RTO and RPO needs.

Test disaster recovery implementation to validate the implementation: Regularly test failover to DR to ensure that RTO and RPO are met.

A pattern to avoid is developing recovery paths that are rarely executed. For example, you might have a secondary data store that is used for read-only queries. When you write to a data store and the primary fails, you might want to fail over to the secondary data store. If you don't frequently test this failover, you might find that your assumptions about the capabilities of the secondary data store are incorrect. The capacity of the secondary, which might have been sufficient when you last tested, may be no longer be able to tolerate the load under this scenario. Our experience has shown that the only error recovery that works is the path you test frequently. This is why having a small number of recovery paths is best. You can establish recovery patterns and regularly test them. If you have a complex or critical recovery path, you still need to regularly execute that failure in production to convince yourself that the recovery path works. In the example we just discussed, you should fail over to the standby regularly, regardless of need.

Manage configuration drift at the DR site or region: Ensure that your infrastructure, data, and configuration are as needed at the DR site or region. For example, check that AMIs and service quotas are up to date.

AWS Config continuously monitors and records your AWS resource configurations. It can detect drift and trigger AWS Systems Manager Automation to fix it and raise alarms. AWS CloudFormation can additionally detect drift in stacks you have deployed.

Automate recovery: Use AWS or third-party tools to automate system recovery and route traffic to the DR site or region.

Based on configured health checks, AWS services, such as Elastic Load Balancing and AWS Auto Scaling, can distribute load to healthy Availability Zones while services, such as Amazon Route 53 and AWS Global Accelerator, can route load to healthy AWS Regions. Amazon Route 53 Application Recovery Controller helps you manage and coordinate failover using readiness check and routing control features. These features continually monitor your application's ability to recover from failures, so you can control application recovery across multiple AWS Regions, Availability Zones, and on premises.

For workloads on existing physical or virtual data centers or private clouds, CloudEndure Disaster Recovery, available through AWS Marketplace, enables organizations to set up an automated disaster recovery strategy to AWS. CloudEndure also supports cross-Region / cross-AZ disaster recovery in AWS.

Resources

Videos

- AWS re:Invent 2019: Backup-and-restore and disaster-recovery solutions with AWS (STG208)

Documentation

- What is AWS Backup?
- What is Amazon Route 53 Application Recovery Controller?
- Remediating Noncompliant AWS Resources by AWS Config Rules
- AWS Systems Manager Automation
- AWS CloudFormation: Detect Drift on an Entire CloudFormation Stack
- Amazon RDS: Cross-region backup copy
- RDS: Replicating a Read Replica Across Regions
- S3: Cross-Region Replication
- Route 53: Configuring DNS Failover
- CloudEndure Disaster Recovery
- How do I implement an Infrastructure Configuration Management solution on AWS?
- CloudEndure Disaster Recovery to AWS
- AWS Marketplace: products that can be used for disaster recovery
- APN Partner: partners that can help with disaster recovery

Example Implementations for Availability Goals

In this section, we'll review workload designs using the deployment of a typical web application that consists of a reverse proxy, static content on Amazon S3, an application server, and a SQL database for persistent storage of data. For each availability target, we provide an example implementation. This workload could instead use containers or AWS Lambda for compute and NoSQL (such as Amazon DynamoDB) for the database, but the approaches are similar. In each scenario, we demonstrate how to meet availability goals through workload design for these topics:

Topic	For more information, see this section
Monitor resources	Monitor Workload Resources (p. 24)
Adapt to changes in demand	Design your Workload to Adapt to Changes in Demand (p. 27)
Implement change	Implement Change (p. 29)
Back up data	Back up Data (p. 33)
Architect for resiliency	Use Fault Isolation to Protect Your Workload (p. 35) Design your Workload to Withstand Component Failures (p. 40)
Test resiliency	Test Reliability (p. 43)
Plan for disaster recovery (DR)	Plan for Disaster Recovery (DR) (p. 46)

Dependency Selection

We have chosen to use Amazon EC2 for our applications. We will show how using Amazon RDS and multiple Availability Zones improves the availability of our applications. We will use Amazon Route 53 for DNS. When we use multiple Availability Zones, we will use Elastic Load Balancing. Amazon S3 is used for backups and static content. As we design for higher reliability, we must use services with higher availability themselves. See Appendix A: Designed-For Availability for Select AWS Services (p. 68) for the design goals for the respective AWS services.

Single-Region Scenarios

Topics

- 2 9s (99%) Scenario (p. 51)
- 3 9s (99.9%) Scenario (p. 52)
- 4 9s (99.99%) Scenario (p. 54)

2 9s (99%) Scenario

These workloads are helpful to the business, but it's only an *inconvenience* if they are unavailable. This type of workload can be internal tooling, internal knowledge management, or project tracking. Or these can be actual customer-facing workloads but served from an experimental service, with a feature toggle that can hide the service if needed.

These workloads can be deployed with one Region and one Availability Zone.

Monitor resources

We will have simple monitoring, indicating whether the service home page is returning an HTTP 200 OK status. When problems occur, our playbook will indicate that logging from the instance will be used to establish root cause.

Adapt to changes in demand

We will have playbooks for common hardware failures, urgent software updates, and other disruptive changes.

Implement change

We will use AWS CloudFormation to define our infrastructure as code, and specifically to speed up reconstruction in the event of a failure.

Software updates are manually performed using a runbook, with downtime required for the installation and restart of the service. If a problem happens during deployment, the runbook describes how to roll back to the previous version.

Any corrections of the error are done using analysis of logs by the operations and development teams, and the correction is deployed after the fix is prioritized and completed.

Back up data

We will use a vendor or purpose built backup solution to send encrypted backup data to Amazon S3 using a runbook. We will test that the backups work by restoring the data and ensuring the ability to use it on a regular basis using a runbook. We configure versioning on our Amazon S3 objects and remove permissions for deletion of the backups. We use an Amazon S3 bucket lifecycle policy to archive or permanently delete according to our requirements.

Architect for resiliency

Workloads are deployed with one Region and one Availability Zone. We deploy the application, including the database, to a single instance.

Test resiliency

The deployment pipeline of new software is scheduled, with some unit testing, but mostly white-box/black-box testing of the assembled workload.

Plan for disaster recovery (DR)

During failures we wait for the failure to finish, optionally routing requests to a static website using DNS modification via a runbook. The recovery time for this will be determined by the speed at which the infrastructure can be deployed and the database restored to the most recent backup. This deployment can either be into the same Availability Zone, or into a different Availability Zone, in the event of an Availability Zone failure, using a runbook.

Availability design goal

We take 30 minutes to understand and decide to execute recovery, deploy the whole stack in AWS CloudFormation in 10 minutes, assume that we deploy to a new Availability Zone, and assume that the database can be restored in 30 minutes. This implies that it takes about 70 minutes to recover from a failure. Assuming one failure per quarter, our estimated impact time for the year is 280 minutes, or four hours and 40 minutes.

This means that the upper limit on availability is 99.9%. The actual availability also depends on the real rate of failure, the duration of failure, and how quickly each failure actually recovers. For this architecture, we require the application to be offline for updates (estimating 24 hours per year: four hours per change, six times per year), plus actual events. So referring to the table on application availability earlier in the whitepaper we see that our **availability design goal** is 99%.

Summary

Topic	Implementation
Monitor resources	Site health check only; no alerting.
Adapt to changes in demand	Vertical scaling via re-deployment.
Implement change	Runbook for deploy and rollback.
Back up data	Runbook for backup and restore.
Architect for resiliency	Complete rebuild; restore from backup.
Test resiliency	Complete rebuild; restore from backup.
Plan for disaster recovery (DR)	Encrypted backups, restore to different Availability Zone if needed.

3 9s (99.9%) Scenario

The next availability goal is for applications for which it's important to be highly available, but they can tolerate short periods of unavailability. This type of workload is typically used for internal operations that have an effect on employees when they are down. This type of workload can also be customer-facing, but are not high revenue for the business and can tolerate a longer recovery time or recovery point. Such workloads include administrative applications for account or information management.

We can improve availability for workloads by using two Availability Zones for our deployment and by separating the applications to separate tiers.

Monitor resources

Monitoring will be expanded to alert on the availability of the website over all by checking for an HTTP 200 OK status on the home page. In addition, there will be alerting on every replacement of a web server and when the database fails over. We will also monitor the static content on Amazon S3 for availability and alert if it becomes unavailable. Logging will be aggregated for ease of management and to help in root cause analysis.

Adapt to changes in demand

Automatic scaling is configured to monitor CPU utilization on EC2 instances, and add or remove instances to maintain the CPU target at 70%, but with no fewer than one EC2 instance per Availability

Zone. If load patterns on our RDS instance indicate that scale up is needed, we will change the instance type during a maintenance window.

Implement change

The infrastructure deployment technologies remain the same as the previous scenario.

Delivery of new software is on a fixed schedule of every two to four weeks. Software updates will be automated, not using canary or blue/green deployment patterns, but rather, using replace in place. The decision to roll back will be made using the runbook.

We will have playbooks for establishing root cause of problems. After the root cause has been identified, the correction for the error will be identified by a combination of the operations and development teams. The correction will be deployed after the fix is developed.

Back up data

Backup and restore can be done using Amazon RDS. It will be executed regularly using a runbook to ensure that we can meet recovery requirements.

Architect for resiliency

We can improve availability for applications by using two Availability Zones for our deployment and by separating the applications to separate tiers. We will use services that work across multiple Availability Zones, such as Elastic Load Balancing, Auto Scaling and Amazon RDS Multi-AZ with encrypted storage via AWS Key Management Service. This will ensure tolerance to failures on the resource level and on the Availability Zone level.

The load balancer will only route traffic to healthy application instances. The health check needs to be at the data plane/application layer indicating the capability of the application on the instance. This check should not be against the control plane. A health check URL for the web application will be present and configured for use by the load balancer and Auto Scaling, so that instances that fail are removed and replaced. Amazon RDS will manage the active database engine to be available in the second Availability Zone if the instance fails in the primary Availability Zone, then repair to restore to the same resiliency.

After we have separated the tiers, we can use distributed system resiliency patterns to increase the reliability of the application so that it can still be available even when the database is temporarily unavailable during an Availability Zone failover.

Test resiliency

We do functional testing, same as in the previous scenario. We do not test the self-healing capabilities of ELB, automatic scaling, or RDS failover.

We will have playbooks for common database problems, security-related incidents, and failed deployments.

Plan for disaster recovery (DR)

Runbooks exist for total workload recovery and common reporting. Recovery uses backups stored in the same region as the workload.

Availability design goal

We assume that at least some failures will require a manual decision to execute recovery. However with the greater automation in this scenario, we assume that only two events per year will require this decision. We take 30 minutes to decide to execute recovery, and assume that recovery is completed within 30 minutes. This implies 60 minutes to recover from failure. Assuming two incidents per year, our estimated impact time for the year is 120 minutes.

This means that the upper limit on availability is 99.95%. The actual availability also depends on the real rate of failure, the duration of the failure, and how quickly each failure actually recovers. For this architecture, we require the application to be briefly offline for updates, but these updates are automated. We estimate 150 minutes per year for this: 15 minutes per change, 10 times per year. This adds up to 270 minutes per year when the service is not available, so our **availability design goal** is 99.9%.

Summary

Topic	Implementation
Monitor resources	Site health check only; alerts sent when down.
Adapt to changes in demand	ELB for web and automatic scaling application tier; resizing Multi-AZ RDS.
Implement change	Automated deploy in place and runbook for rollback.
Back up data	Automated backups via RDS to meet RPO and runbook for restoring.
Architect for resiliency	Automatic scaling to provide self-healing web and application tier; RDS is Multi-AZ.
Test resiliency	ELB and application are self-healing; RDS is Multi-AZ; no explicit testing.
Plan for disaster recovery (DR)	Encrypted backups via RDS to same AWS Region.

4 9s (99.99%) Scenario

This availability goal for applications requires the application to be highly available and tolerant to component failures. The application must be able to absorb failures without needing to get additional resources. This availability goal is for mission critical applications that are main or significant revenue drivers for a corporation, such as an ecommerce site, a business to business web service, or a high traffic content/media site.

We can improve availability further by using an architecture that will be *statically stable* within the Region. This availability goal doesn't require a control plane change in behavior of our workload to tolerate failure. For example, there should be enough capacity to withstand the loss of one Availability Zone. We should not require updates to Amazon Route 53 DNS. We should not need to create any new infrastructure, whether it's creating or modifying an S3 bucket, creating new IAM policies (or modifications of policies), or modifying Amazon ECS task configurations.

Monitor resources

Monitoring will include success metrics as well as alerting when problems occur. In addition, there will be alerting on every replacement of a failed web server, when the database fails over, and when an AZ fails.

Adapt to changes in demand

We will use Amazon Aurora as our RDS, which enables automatic scaling of read replicas. For these applications, engineering for read availability over write availability of primary content is also a key architecture decision. Aurora can also automatically grow storage as needed, in 10 GB increments up to 64 TB.

Implement change

We will deploy updates using canary or blue/green deployments into each isolation zone separately. The deployments are fully automated, including a roll back if KPIs indicate a problem.

Runbooks will exist for rigorous reporting requirements and performance tracking. If successful operations are trending toward failure to meet performance or availability goals, a playbook will be used to establish what is causing the trend. Playbooks will exist for undiscovered failure modes and security incidents. Playbooks will also exist for establishing the root cause of failures. We will also engage with AWS Support for Infrastructure Event Management offering.

The team that builds and operates the website will identify the correction of error of any unexpected failure and prioritize the fix to be deployed after it is implemented.

Back up data

Backup and restore can be done using Amazon RDS. It will be executed regularly using a runbook to ensure that we can meet recovery requirements.

Architect for resiliency

We recommend three Availability Zones for this approach. Using a three Availability Zone deployment, each AZ has static capacity of 50% of peak. Two Availability Zones could be used, but the cost of the statically stable capacity would be more because both zones would have to have 100% of peak capacity. We will add Amazon CloudFront to provide geographic caching, as well as request reduction on our application's data plane.

We will use Amazon Aurora as our RDS and deploy read replicas in all three zones.

The application will be built using the software/application resiliency patterns in all layers.

Test resiliency

The deployment pipeline will have a full test suite, including performance, load, and failure injection testing**.**

We will practice our failure recovery procedures constantly through game days, using runbooks to ensure that we can perform the tasks and not deviate from the procedures. The team that builds the website also operates the website.

Plan for disaster recovery (DR)

Runbooks exist for total workload recovery and common reporting. Recovery uses backups stored in the same region as the workload. Restore procedures are regularly exercised as part of game days.

Availability design goal

We assume that at least some failures will require a manual decision to execute recovery, however with greater automation in this scenario we assume that only two events per year will require this decision and the recovery actions will be rapid. We take 10 minutes to decide to execute recovery, and assume that recovery is completed within five minutes. This implies 15 minutes to recover from failure. Assuming two failures per year, our estimated impact time for the year is 30 minutes.

This means that the upper limit on availability is 99.99%. The actual availability will also depend on the real rate of failure, the duration of the failure, and how quickly each failure actually recovers. For this architecture, we assume that the application is online continuously through updates. Based on this, our **availability design goal** is 99.99%.

Summary

Topic	Implementation
Monitor resources	Health checks at all layers and on KPIs; alerts sent when configured alarms are tripped; alerting on all failures. Operational meetings are rigorous to detect trends and manage to design goals.
Adapt to changes in demand	ELB for web and automatic scaling application tier; automatic scaling storage and read replicas in multiple zones for Aurora RDS.
Implement change	Automated deploy via canary or blue/green and automated rollback when KPIs or alerts indicate undetected problems in application. Deployments are made by isolation zone.
Back up data	Automated backups via RDS to meet RPO and automated restoration that is practiced regularly in a game day.
Architect for resiliency	Implemented fault isolation zones for the application; auto scaling to provide self-healing web and application tier; RDS is Multi-AZ.
Test resiliency	Component and isolation zone fault testing is in pipeline and practiced with operational staff regularly in a game day; playbooks exist for diagnosing unknown problems; and a Root Cause Analysis process exists.
Plan for disaster recovery (DR)	Encrypted backups via RDS to same AWS Region that is practiced in a game day.

Multi-Region Scenarios

Implementing our application in multiple AWS Regions will increase the cost of operation, partly because we isolate regions to maintain their autonomy. It should be a very thoughtful decision to pursue this path. That said, regions provide a strong isolation boundary and we take great pains to avoid correlated failures across regions. Using multiple regions will give you greater control over your recovery time in the event of a hard dependency failure on a regional AWS service. In this section, we'll discuss various implementation patterns and their typical availability.

Topics

- 3½ 9s (99.95%) with a Recovery Time between 5 and 30 Minutes (p. 56)
- 5 9s (99.999%) or Higher Scenario with a Recovery Time under 1 minute (p. 59)

3½ 9s (99.95%) with a Recovery Time between 5 and 30 Minutes

This availability goal for applications requires extremely short downtime and very little data loss during specific times. Applications with this availability goal include applications in the areas of: banking,

investing, emergency services, and data capture. These applications have very short recovery times and recovery points.

We can improve recovery time further by using a *Warm Standby* approach across two AWS Regions. We will deploy the entire workload to both Regions, with our passive site scaled down and all data kept eventually consistent. Both deployments will be *statically stable* within their respective regions. The applications should be built using the distributed system resiliency patterns. We'll need to create a lightweight *routing* component that monitors for workload health, and can be configured to route traffic to the passive region if necessary.

Monitor resources

There will be alerting on every replacement of a web server, when the database fails over, and when the Region fails over. We will also monitor the static content on Amazon S3 for availability and alert if it becomes unavailable. Logging will be aggregated for ease of management and to help in root cause analysis in each Region.

The routing component monitors both our application health and any regional hard dependencies we have.

Adapt to changes in demand

Same as the 4 9s scenario.

Implement change

Delivery of new software is on a fixed schedule of every two to four weeks. Software updates will be automated using canary or blue/green deployment patterns.

Runbooks exist for when Region failover occurs, for common customer issues that occur during those events, and for common reporting.

We will have playbooks for common database problems, security-related incidents, failed deployments, unexpected customer issues on Region failover, and establishing root cause of problems. After the root cause has been identified, the correction of error will be identified by a combination of the operations and development teams and deployed when the fix is developed.

We will also engage with AWS Support for Infrastructure Event Management.

Back up data

Like the 4 9s scenario, we automatic RDS backups and use S3 versioning. Data is automatically and asynchronously replicated from the Aurora RDS cluster in the active region to cross-region read replicas in the passive region. S3 cross-region replication is used to automatically and asynchronously move data from the active to the passive region.

Architect for resiliency

Same as the 4 9s scenario, plus regional failover is possible. This is managed manually. During failover, we will route requests to a static website using DNS failover until recovery in the second Region.

Test resiliency

Same as the 4 9s scenario plus we will validate the architecture through game days using runbooks. Also RCA correction is prioritized above feature releases for immediate implementation and deployment

Plan for disaster recovery (DR)

Regional failover is manually managed. All data is asynchronously replicated. Infrastructure in the *warm standby* is scaled out. This can be automated using a workflow executed on AWS Step Functions. AWS Systems Manager (SSM) can also help with this automation, as you can create SSM documents that update Auto Scaling groups and resize instances.

Availability design goal

We assume that at least some failures will require a manual decision to execute recovery, however with good automation in this scenario we assume that only two events per year will require this decision. We take 20 minutes to decide to execute recovery, and assume that recovery is completed within 10 minutes. This implies that it takes 30 minutes to recover from failure. Assuming two failures per year, our estimated impact time for the year is 60 minutes.

This means that the upper limit on availability is 99.95%. The actual availability will also depend on the real rate of failure, the duration of the failure, and how quickly each failure actually recovers. For this architecture, we assume that the application is online continuously through updates. Based on this, our **availability design goal** is 99.95%.

Summary

Topic	Implementation
Monitor resources	Health checks at all layers, including DNS health at AWS Region level, and on KPIs; alerts sent when configured alarms are tripped; alerting on all failures. Operational meetings are rigorous to detect trends and manage to design goals.
Adapt to changes in demand	ELB for web and automatic scaling application tier; automatic scaling storage and read replicas in multiple zones in the active and passive regions for Aurora RDS. Data and infrastructure synchronized between AWS Regions for static stability.
Implement change	Automated deploy via canary or blue/green and automated rollback when KPIs or alerts indicate undetected problems in application, deployments are made to one isolation zone in one AWS Region at a time.
Back up data	Automated backups in each AWS Region via RDS to meet RPO and automated restoration that is practiced regularly in a game day. Aurora RDS and S3 data is automatically and asynchronously replicated from active to passive region.
Architect for resiliency	Automatic scaling to provide self-healing web and application tier; RDS is Multi-AZ; regional failover is managed manually with static site presented while failing over.
Test resiliency	Component and isolation zone fault testing is in pipeline and practiced with operational staff regularly in a game day; playbooks exist for

Topic	Implementation
	diagnosing unknown problems; and a Root Cause Analysis process exists, with communication paths for what the problem was, and how it was corrected or prevented. RCA correction is prioritized above feature releases for immediate implementation and deployment.
Plan for disaster recovery (DR)	Warm Standby deployed in another region. Infrastructure is scaled out using workflows executed using AWS Step Functions or AWS Systems Manager Documents. Encrypted backups via RDS. Cross-region read replicas between two AWS Regions. Cross-region replication of static assets in Amazon S3. Restoration is to the current active AWS Region, is practiced in a game day, and is coordinated with AWS.

5 9s (99.999%) or Higher Scenario with a Recovery Time under 1 minute

This availability goal for applications requires almost no downtime or data loss for specific times. Applications that could have this availability goal include, for example certain banking, investing, finance, government, and critical business applications that are the core business of an extremely large-revenue generating business. The desire is to have strongly consistent data stores and complete redundancy at all layers. We have selected a SQL-based data store. However, in some scenarios, we will find it difficult to achieve a very small RPO. If you can partition your data, it's possible to have no data loss. This might require you to add application logic and latency to ensure that you have consistent data between geographic locations, as well as the capability to move or copy data between partitions. Performing this partitioning might be easier if you use a NoSQL database.

We can improve availability further by using an *Active-Active* approach across multiple AWS Regions. The workload will be deployed in all desired Regions that are *statically stable* across regions (so the remaining regions can handle load with the loss of one region). A *routing* layer directs traffic to geographic locations that are healthy and automatically changes the destination when a location is unhealthy, as well as temporarily stopping the data replication layers. Amazon Route 53 offers 10-second interval health checks and also offers TTL on your record sets as low as one second.

Monitor resources

Same as the 3½ 9s scenario, plus alerting when a Region is detected as unhealthy, and traffic is routed away from it.

Adapt to changes in demand

Same as the 3½ 9s scenario.

Implement change

The deployment pipeline will have a full test suite, including performance, load, and failure injection testing. We will deploy updates using canary or blue/green deployments to one isolation zone at a time, in one Region before starting at the other. During the deployment, the old versions will still be kept

running on instances to facilitate a faster rollback. These are fully automated, including a rollback if KPIs indicate a problem. Monitoring will include success metrics as well as alerting when problems occur.

Runbooks will exist for rigorous reporting requirements and performance tracking. If successful operations are trending towards failure to meet performance or availability goals, a playbook will be used to establish what is causing the trend. Playbooks will exist for undiscovered failure modes and security incidents. Playbooks will also exist for establishing root cause of failures.

The team that builds the website also operates the website. That team will identify the correction of error of any unexpected failure and prioritize the fix to be deployed after it's implemented. We will also engage with AWS Support for Infrastructure Event Management.

Back up data

Same as the 3½ 9s scenario.

Architect for resiliency

The applications should be built using the software/application resiliency patterns. It's possible that many other routing layers may be required to implement the needed availability. The complexity of this additional implementation should not be underestimated. The application will be implemented in deployment fault isolation zones, and partitioned and deployed such that even a Region wide-event will not affect all customers.

Test resiliency

We will validate the architecture through game days using runbooks to ensure that we can perform the tasks and not deviate from the procedures.

Plan for disaster recovery (DR)

Active-Active multi-region deployment with complete workload infrastructure and data in multiple regions. Using a read local, write global strategy, one region is the primary database for all writes, and data is replicated for reads to other regions. If the primary DB region fails, a new DB will need to be promoted. Read local, write global has users assigned to a home region where DB writes are handled. This lets users read or write from any region, but requires complex logic to manage potential data conflicts across writes in different regions.

When a region is detected as unhealthy, the routing layer automatically routes traffic to the remaining healthy regions. No manual intervention is required.

Data stores must be replicated between the Regions in a manner that can resolve potential conflicts. Tools and automated processes will need to be created to copy or move data between the partitions for latency reasons and to balance requests or amounts of data in each partition. Remediation of the data conflict resolution will also require additional operational runbooks.

Availability design goal

We assume that heavy investments are made to automate all recovery, and that recovery can be completed within one minute. We assume no manually triggered recoveries, but up to one automated recovery action per quarter. This implies four minutes per year to recover. We assume that the application is online continuously through updates. Based on this, our **availability design goal** is 99.999%.

Summary

Topic	Implementation
Monitor resources	Health checks at all layers, including DNS health at AWS Region level, and on KPIs; alerts sent when configured alarms are tripped; alerting on all failures. Operational meetings are rigorous to detect trends and manage to design goals.
Adapt to changes in demand	ELB for web and automatic scaling application tier; automatic scaling storage and read replicas in multiple zones in the active and passive regions for Aurora RDS. Data and infrastructure synchronized between AWS Regions for static stability.
Implement change	Automated deploy via canary or blue/green and automated rollback when KPIs or alerts indicate undetected problems in application, deployments are made to one isolation zone in one AWS Region at a time.
Back up data	Automated backups in each AWS Region via RDS to meet RPO and automated restoration that is practiced regularly in a game day. Aurora RDS and S3 data is automatically and asynchronously replicated from active to passive region.
Architect for resiliency	Implemented fault isolation zones for the application; auto scaling to provide self-healing web and application tier; RDS is Multi-AZ; regional failover automated.
Test resiliency	Component and isolation zone fault testing is in pipeline and practiced with operational staff regularly in a game day; playbooks exist for diagnosing unknown problems; and a Root Cause Analysis process exists with communication paths for what the problem was, and how it was corrected or prevented. RCA correction is prioritized above feature releases for immediate implementation and deployment.
Plan for disaster recovery (DR)	Active-Active deployed across at least two regions. Infrastructure is fully scaled and statically stable across regions. Data is partitioned and synchronized across regions. Encrypted backups via RDS. Region failure is practiced in a game day, and is coordinated with AWS. During restoration a new database primary may need to be promoted.

Resources

Documentation

- The Amazon Builders' Library - How Amazon builds and operates software

- AWS Architecture Center

Labs

- AWS Well-Architected Reliability Labs

External Links

- Adaptive Queuing Pattern: Fail at Scale
- Availability and Beyond: Understanding and improving the resilience of distributed systems on AWS

Books

- Robert S. Hammer "Patterns for Fault Tolerant Software"
- Andrew Tanenbaum and Marten van Steen "Distributed Systems: Principles and Paradigms"

Conclusion

Whether you are new to the topics of availability and reliability, or a seasoned veteran seeking insights to maximize your mission critical workload's availability, we hope this whitepaper has triggered your thinking, offered a new idea, or introduced a new line of questioning. We hope this leads to a deeper understanding of the right level of availability based on the needs of your business, and how to design the reliability to achieve it. We encourage you to take advantage of the design, operational, and recovery-oriented recommendations offered here as well as the knowledge and experience of our AWS Solution Architects. We'd love to hear from you–especially about your success stories achieving high levels of availability on AWS. Contact your account team or use Contact US on our website.

Contributors

Contributors to this document include:

- Seth Eliot, Principal Reliability Solutions Architect Well-Architected, Amazon Web Services
- Adrian Hornsby, Principal Technical Evangelist Architecture, Amazon Web Services
- Rodney Lester, Principal Solutions Architect, Amazon Web Services
- Kevin Miller, Director Software Development, Amazon Web Services
- Shannon Richards, Sr. Technical Program Manager, Amazon Web Services
- Aden Leirer, Content Program Manager Well-Architected, Amazon Web Services

Further Reading

For additional information, see:

- AWS Well-Architected Framework
- AWS Architecture Center

Document Revisions

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

update-history-change	update-history-description	update-history-date
Minor update (p. 1)	Added Sustainability Pillar to introduction.	December 2, 2021
Minor update (p. 66)	Added information about AWS Fault Injection Simulator (AWS FIS).	March 15, 2021
Minor update (p. 66)	Minor text update.	January 4, 2021
Whitepaper updated (p. 66)	Updated Appendix A to update the Availability Design Goal for Amazon SQS, Amazon SNS, and Amazon MQ; Re-order rows in table for easier lookup; Improve explanation of differences between availability and disaster recovery and how they both contribute to resiliency; Expand coverage of multi-region architectures (for availability) and multi-region strategies (for disaster recovery); Update referenced book to latest version; Expand availability calculations to include request-based calculation, and shortcut calculations; Improve description for Game Days	December 7, 2020
Minor update (p. 66)	Updated Appendix A to update the Availability Design Goal for AWS Lambda	October 27, 2020
Minor update (p. 66)	Updated Appendix A to add the Availability Design Goal for AWS Global Accelerator	July 24, 2020
Updates for new Framework (p. 66)	Substantial updates and new/revised content, including: Added “Workload Architecture” best practices section, re-organized best practices into Change Management and Failure Management sections, updated Resources, updated to include latest AWS resources and services such as AWS Global Accelerator, AWS Service Quotas, and AWS Transit Gateway,	July 8, 2020

	added/updated definitions for Reliability, Availability, Resiliency, better aligned whitepaper to the AWS Well-Architected Tool (questions and best practices) used for Well-Architected Reviews, re-order design principles, moving **Automatically recover from failure** before **Test recovery procedures**, updated diagrams and formats for equations, removed Key Services sections and instead integrated references to key AWS services into the best practices.	
Minor update (p. 66)	Fixed broken link	October 1, 2019
Whitepaper updated (p. 66)	Appendix A updated	April 1, 2019
Whitepaper updated (p. 66)	Added specific AWS Direct Connect networking recommendations and additional service design goals	September 1, 2018
Whitepaper updated (p. 66)	Added Design Principles and Limit Management sections. Updated links, removed ambiguity of upstream/ downstream terminology, and added explicit references to the remaining Reliability Pillar topics in the availability scenarios.	June 1, 2018
Whitepaper updated (p. 66)	Changed DynamoDB Cross Region solution to DynamoDB Global Tables. Added service design goals	March 1, 2018
Minor updates (p. 66)	Minor correction to availability calculation to include application availability	December 1, 2017
Whitepaper updated (p. 66)	Updated to provide guidance on high availability designs, including concepts, best practice and example implementations.	November 1, 2017
Initial publication (p. 66)	Reliability Pillar - AWS Well-Architected Framework published.	November 1, 2016

Appendix A: Designed-For Availability for Select AWS Services

Below, we provide the availability that select AWS services were designed to achieve. These values do not represent a Service Level Agreement or guarantee, but rather provide insight to the design goals of each service. In certain cases, we differentiate portions of the service where there's a meaningful difference in the availability design goal. This list is not comprehensive for all AWS services, and we expect to periodically update with information about additional services. Amazon CloudFront, Amazon Route 53, AWS Global Accelerator, and the AWS Identity and Access Management Control Plane provide global service, and the component availability goal is stated accordingly. Other services provide services within an AWS Region and the availability goal is stated accordingly. Many services operate within an Availability Zone, separate from those in other Availability Zones. In these cases, we provide the availability design goal for a single AZ, and when any two (or more) Availability Zones are used.

Note
The numbers in the following table do not refer to durability (long term retention of data); they are availability numbers (access to data or functions.)

Service	Component	Availability Design Goal
Amazon API Gateway	Control Plane	99.950%
	Data Plane	99.990%
Amazon Aurora	Control Plane	99.950%
	Single-AZ Data Plane	99.950%
	Multi-AZ Data Plane	99.990%
Amazon CloudFront	Control Plane	99.900%
	Data Plane (content delivery)	99.990%
Amazon CloudSearch	Control Plane	99.950%
	Data Plane	99.950%
Amazon CloudWatch	CW Metrics (service)	99.990%
	CW Events (service)	99.990%
	CW Logs (service)	99.950%
Amazon DynamoDB	Service (standard)	99.990%
	Service (Global Tables)	99.999%
Amazon Elastic Block Store	Control Plane	99.950%
	Data Plane (volume availability)	99.999%
Amazon Elastic Compute Cloud (Amazon EC2)	Control Plane	99.950%
	Single-AZ Data Plane	99.950%

Service	Component	Availability Design Goal
	Multi-AZ Data Plane	99.990%
Amazon Elastic Container Service (Amazon ECS)	Control Plane	99.900%
	EC2 Container Registry	99.990%
	EC2 Container Service	99.990%
Amazon Elastic File System	Control Plane	99.950%
	Data Plane	99.990%
Amazon ElastiCache	Service	99.990%
Amazon OpenSearch Service	Control Plane	99.950%
	Data Plane	99.950%
Amazon EMR	Control Plane	99.950%
Amazon Kinesis Data Firehose	Service	99.900%
Amazon Kinesis Data Streams	Service	99.990%
Amazon Kinesis Video Streams	Service	99.900%
Amazon Managed Streaming for Apache Kafka (Amazon MSK)	Control Plane	99.950%
	Three-AZ Data Plane	99.990%
	Two-AZ Data Plane	99.950%
Amazon MQ	Data Plane	99.950%
	Control Plane	99.950%
Amazon Neptune	Service	99.900%
Amazon Redshift	Control Plane	99.950%
	Data Plane	99.950%
Amazon Rekognition	Service	99.980%
Amazon Relational Database Service (Amazon RDS)	Control Plane	99.950%
	Single-AZ Data Plane	99.950%
	Multi-AZ Data Plane	99.990%
Amazon Route 53	Control Plane	99.950%
	Data Plane (query resolution)	100.000%
Amazon SageMaker	Data Plane (Model Hosting)	99.990%
	Control Plane	99.950%

Service	Component	Availability Design Goal
Amazon Simple Notification Service (Amazon SNS)	Data Plane	99.990%
	Control Plane	99.900%
Amazon Simple Queue Service (Amazon SQS)	Data Plane	99.980%
	Control Plane	99.900%
Amazon Simple Storage Service (Amazon S3)	Service (Standard)	99.990%
Amazon S3 Glacier	Service	99.900%
AWS Auto Scaling	Control Plane	99.900%
	Data Plane	99.990%
AWS Batch	Control Plane	99.900%
	Data Plane	99.950%
AWS CloudFormation	Service	99.950%
AWS CloudHSM	Control Plane	99.900%
	Single-AZ Data Plane	99.900%
	Multi-AZ Data Plane	99.990%
AWS CloudTrail	Control Plane (config)	99.900%
	Data Plane (data events)	99.990%
	Data Plane (management events)	99.999%
AWS Config	Service	99.950%
AWS Data Pipeline	Service	99.990%
AWS Database Migration Service (AWS DMS)	Control Plane	99.900%
	Data Plane	99.950%
AWS Direct Connect	Control Plane	99.900%
	Single Location Data Plane	99.900%
	Multi Location Data Plane	99.990%
AWS Global Accelerator	Control Plane	99.900%
	Single-Network Zone Data Plane	99.950%
	Two-Network Zone Data Plane	99.995%
AWS Glue	Service	99.990%

Service	Component	Availability Design Goal
AWS Identity and Access Management	Control Plane	99.900%
	Data Plane (authentication)	99.995%
AWS IoT Core	Service	99.900%
AWS IoT Device Management	Service	99.900%
AWS IoT Greengrass	Service	99.900%
AWS Key Management Service (AWS KMS)	Control Plane	99.990%
	Data Plane	99.995%
AWS Lambda	Function Invocation	99.990%
AWS Secrets Manager	Service	99.900%
AWS Shield	Control Plane	99.500%
	Data Plane (detection)	99.000%
	Data Plane (mitigation)	99.900%
AWS Storage Gateway	Control Plane	99.950%
	Data Plane	99.950%
AWS X-Ray	Control Plane (console)	99.900%
	Data Plane	99.950%
Elastic Load Balancing	Control Plane	99.950%
	Data Plane	99.990%