Reliability Pillar

AWS Well-Architected Framework

July 2018

Notices

Contents

Abstract

The focus of this paper is the reliability pillar of the AWS Well-Architected Framework. It provides guidance to help you 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 systems on AWS. By using the Framework you will learn architectural best practices for designing and operating reliable, secure, efficient, and cost-effective systems 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 systems greatly increases the likelihood of business success.

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

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

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, 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 system to recover from infrastructure or service disruptions, dynamically acquire computing resources to meet demand, and mitigate disruptions such as misconfigurations or transient network issues. This paper provides in-depth, best-practice guidance for architecting reliable systems on AWS.

Design Principles

In the cloud, there are a number of principles that can help you increase reliability:

- **Test recovery procedures:** In an on-premises environment, testing is often conducted to prove the system works in a particular scenario; testing is not typically used to validate recovery strategies. In the cloud, you can test how your system 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 exposes failure pathways that you can test and fix *before* a real failure scenario, reducing the risk of components that have not been tested before failing.
- **Automatically recover from failure:** By monitoring a system 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 is possible to anticipate and remediate failures before they occur.
- **Scale horizontally to increase aggregate system availability:** Replace one large resource with multiple small resources to reduce the impact of a single failure on the overall system. 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 systems is resource saturation, when the demands placed on a system exceed the capacity of that system (this is often the objective of denial of service attacks). In the cloud, you can monitor demand and system 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 limits can be controlled and others can be managed (See Foundation-Limit Management).
- **Manage change in automation**: Changes to you infrastructure should be via automation. The changes that need to be managed are changes to the automation.

We will discuss all these design principals when illustrating scenarios.

Definition

Service availability is commonly defined as the percentage of time that an application is operating normally. That is, it's the percentage of time that it's correctly performing the operations expected of it. This percentage is calculated over periods of time, such as a month, year, or trailing 3 years. Applying the strictest possible interpretation, availability is reduced any time the application isn't operating normally, including both scheduled and unscheduled interruptions. We define *availability* using the following criteria:

- Availability = Normal Operation Time / Total Time
- A percentage of uptime (such as 99.9%) over a period of time (commonly a year)
- Common short-hand refers only to the "number of 9's"; for example, "five nines" translates to 99.999% available
- Some customers choose to exclude scheduled service downtime (for example, planned maintenance) from the Total Time in the formula in the first bullet. However, this is often a false choice because customers might actually want to use your service during these times.

Here is a table of common application availability design goals and the possible length of interruptions that 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. In this document, we will refer to these values.

Availability	Max Disruption (per year)	Application Categories
99%	3 days 15 hours	Batch processing, data extraction, transfer, and load jobs
99.9%	8 hours 45 minutes	Internal tools like knowledge management, project tracking
99.95%	4 hours 22 minutes	Online commerce, point of sale
99.99%	52 minutes	Video delivery, broadcast systems
99.999%	5 minutes	ATM transactions, telecommunications systems

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 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 system can theoretically achieve 99.97% availability:

invoking system * dependent 1 * dependent 2 =

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 Availability Zones), the theoretical availability is computed as 100% minus the product of the component failure rates (100% minus availability.) For example, if a system makes use of two independent components, each with an availability of 99.9%, the resulting system availability is >99.999%:

maximum availability - ((downtime of dependent 1) * (downtime of dependent 2)) =

100% - (0.1% * 0.1%) = 99.9999%

But what if I don't know the availability of a dependency?

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). But in cases where this isn't available (for example, a component where the manufacturer does not publish availability information), one simple 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:

Availability Estimate = 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: This document can help you calculate your availability.

Costs for availability. Designing applications for higher levels of availability typically comes with increased costs, so it's appropriate to identify the true availability needs before embarking on 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 will diminish 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.

Foundation – Limit Management

When architecting systems there are physical limits and resource constraints that need to be taken into account. A common source of failure, and a reason for a lack of availability, is resource constraint. For example, the rate that you can push bits down a fiber optic cable, or the amount of storage on a physical disk. Understanding physical constraints is the first part of designing reliable systems. Second, with service-based architecture, there are often service limits that act to protect the service from breaching Service Level Agreements (rate limits) or design constraints (hard limits). The final piece of limit management is alerting and reporting, which enable you to know when you hit a limit or are about to hit a limit, and then you can react accordingly.

The default limits for cloud resources created by AWS services are documented for each service. These limits are tracked per account, so if you use multiple accounts, you need to know what the limits are in each account. Other limits may be based on your configuration. Examples of these limits are number of instances in an Auto Scaling group, provisioned IOPS, RDS storage allocated, EBS volume allocations, network IO, available IP addresses in a subnet or VPC, etc.

Limits are enforced per AWS Region and per AWS account. If you are planning to deploy into multiple regions or AWS accounts, then you should ensure that you increase limits in the regions and accounts that you using. Additionally, ensure you have sufficient buffer accounted for, such that an Availability Zone event will not cause you to exceed your limits when requesting additional resources while the unhealthy resources are being terminated.

AWS provides a list of some service limits via AWS Trusted Advisor, and others are available from the AWS Management Console. The default service limits that are provided are available in the Service Limits documentation; you can contact AWS Support to provide your current limits for the services you are using if you have not tracked your limit increase requests. For rate limits on throttled APIs, the SDKs provide mechanisms (retry, exponential back off) to handle throttled responses. You should evaluate your use cases to decide which scheme works better for you. If you have a use case where the throttling limits impact your application's performance, then contact AWS Support to see if there are mitigations or if the limit can be increased.

Ideally, limit tracking is automated. You can store what your current service limits are in a persistent data store like Amazon DynamoDB. If you integrate your Configuration Management Database (CMDB) or ticketing system with AWS Support APIs, you can automate the tracking of limit increase requests and current limits. If you integrate with a CMDB, then it is likely that you can store the service limits within that system.

Key AWS Services

The key AWS feature that supports a way to identify what service limits currently are is **AWS Trusted Advisor** which provides a list of what limits it returns. The following services and features are also important:

- **Amazon CloudWatch**: You can set alarms to indicate when you are getting close to limits in Network IO, Provisioned IOPS, EBS and ephemeral volume capacity (through custom metrics), etc. You can also set alarms for when you are approaching maximum capacity of auto scaling groups.
- **Amazon CloudWatch**–Logs: Metric filters can be used to search and extract patterns in a log event. Log entries are converted to numeric metrics, and alarms can be applied.

Resources

Refer to the following resources to learn more about AWS best practices for identifying limits and managing limits, and see AWS Answers for an example of automated limit monitoring:

Video

- How do I manage my AWS service limits?

Documentation

- AWS Service Limits
- Service Limit Reports Blog Post
- Trusted Advisor FAQs
- AWS Limit Monitor on AWS Answers

Foundation - Networking

When architecting systems using IP-address-based networks you need to plan network topology and addressing in anticipation of future growth and integration with other systems and their networks. You might find that your infrastructure is limited if you don’t plan for growth, or you might have difficulties integrating incompatible addressing structures.

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.

When you plan your network topology, the first step is to define the IP address space itself. Following RFC 1918 guidelines, Classless Inter-Domain Routing (CIDR) blocks should be allocated for each VPC. Consider doing the following things as part of this process:

- Allow IP address space for more than one VPC per Region.
- Consider cross-account connections. For example, each line of business might have a unique account and VPCs. These accounts should be able to connect back to shared services.
- Within a VPC, allow space for multiple subnets that span multiple Availability Zones.
- Always leave unused CIDR block space within a VPC.

The second step in planning your network topology is to ensure the resiliency of connectivity:

- 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 Denial of Service (DoS) attack?

AWS has many features that will influence your design. How many VPCs do you plan to use? Will you use Amazon VPC peering between your VPCs? Will you connect virtual private networks (VPNs) to any of these VPCs? Are you going to use AWS Direct Connect or the internet?

A best practice is to always use private address ranges as identified by RFC 1918 for your VPC CIDR blocks. The range you pick should not overlap your existing use or anything that you plan to share address space with using VPC peering or VPN. In general, you need to make sure your allocated range includes sufficient address space for the number of subnets you need to deploy, the potential size of Elastic Load Balancing (ELB) load balancers, the number of concurrent

Lambda invocations within your VPC, and your servers (including machine learning servers) and containers deployed within your subnets. In general, you should plan on deploying large VPC CIDR blocks. Note that VPC CIDR blocks can be changed after they are deployed, but if you allocate large CIDR ranges for your VPC, it will be easier to manage in the long term. Subnet CIDRs cannot be changed. Keep in mind that deploying the largest VPC possible results in over 65,000 IP addresses. The base 10.x.x.x address space means that you can use over 16,000,000 IP addresses. You should err on the side of too large instead of too small for all these decisions.

The connectivity from a VPC is governed through route table entries. An internet gateway, NAT Gateway, virtual private gateway, or VPC peering connection are all exposed to a subnet through an entry in its route table. When you plan your network. Consider the virtual private gateway and VPC peering that you want.

An additional option for inter VPC connectivity is VPC Endpoint Services. This enables you to use a Network Load Balancer as a private entry point from another VPC.

Another option for setting up networking between VPCs is to use VPN appliances. Commonly used appliances are available on the AWS Marketplace.

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. For example, if you choose to connect your VPC to your data center via an AWS Direct Connect connection, you should have a redundant connection fallback either through a second Direct Connect connection from another provider or through the internet. 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 acceptable time to recovery (TTR) and test to ensure you can meet those requirements.

You should use existing standards for protecting your resources within this private address space. A subnet or set of subnets (one per Availability Zone) should be used as a barrier between the internet and your applications. In an on-premises environment, you often have firewalls with features to deflect common web attacks, and you often have load balancers or edge routers to deflect DoS attacks, such as SYN floods. AWS provides many services that can provide this functionality, such as AWS Shield and AWS Shield Advanced, an integrated web application firewall (AWS WAF) deployed on Amazon CloudFront and on ELB, ELB itself, and features of AWS virtual networking like VPC security groups and network access control lists (ACLs). You can augment these features with virtual appliances from AWS Partners and the AWS Marketplace to meet your needs.

Key AWS Services for Network Topology

The key AWS service that supports your network planning is **Amazon Virtual Private Cloud (Amazon VPC)**, which allows you to allocate private IP address ranges to either provide non-internet-accessible resources or to extend your data center. The following services and features are also important:

- **AWS Direct Connect**: Can be used to give a private dedicated connection to AWS for possible lower latency and consistent performance to and from AWS.
- **Amazon EC2**: If you choose to implement VPNs between your networks, this is the service on which you run VPN appliances.
- **Amazon Route 53**: A Domain Name System (DNS) service that is integrated directly with ELB and can help provide a layer of defense in the event of a DoS attack.
- **Elastic Load Balancing**: Provides load balancing across Availability Zones, performs Layer 7 routing, integrates with AWS WAF, and integrates with Auto Scaling to help create a self-healing infrastructure and absorb increases in traffic while releasing resources when traffic decreases.
- **AWS Shield**: Provides automatic protection against Distributed Denial of Service (DDoS) attacks at no extra cost. Additional protection within your provisioned infrastructure is available as AWS Shield Advanced and will protect Elastic Load Balancing load balancers, Amazon CloudFront distributions, and Amazon Route 53-hosted zones.

Resources for Network Topology

Refer to the following resources to learn more about AWS best practices for network planning.

Videos

- Advanced VPC Design and New Capabilities for Amazon VPC (NET305)
- Networking Many VPCs: Transit and Shared Architectures (NET 304)

Documentation

- Amazon Virtual Private Cloud Product Page
- Amazon Virtual Private Cloud Documentation
- Announcement on Amazon VPC allowing customers to expand their VPCs
- VPC Endpoint Services (AWS PrivateLink)
- AWS Global Transit Network on AWS Answers
- Amazon EC2 Instance Types Product Page
- Amazon EC2 Instance Types Documentation
- AWS Marketplace for Network Infrastructure
- AWS Shield Documentation
- AWS Best Practices for DDoS Resiliency Whitepaper
- Single Region Multi-VPC Connectivity on AWS Answers
- Amazon VPC Connectivity Options Whitepaper

Application Design for High Availability

The purpose of this section is to help you think through building and operating applications on AWS with the right level of availability to meet your business needs. Availability goals can vary from those applicable to internal tools (for example, 99% availability) to those for mission critical workloads (for example, 99.999% or even higher.) Based on the necessary availability, the level of effort that's required of engineering and operations, and the services that are appropriate to use to deliver the application will vary. Costs can be considerable to achieve the highest levels of availability. We'll share several practical techniques for applying AWS services to achieve the availability your workloads require.

Note: If the topic of Reliability is new to you, or if you're new to AWS, check out the Automate Deployments to Eliminate Impact (Change Management) and Recovery Oriented Computing (Failure Management) sections later in this whitepaper. These sections will cover some concepts that will be helpful as you read this section.

When designing a new application, it's common to assume that it must be "five nines" (99.999%) available without appreciating the true cost to build and operate applications at that level of availability. Doing so requires that every piece of networking and infrastructure from end customers to the service application, data stores, and all other components must be built and operated to achieve 99.999%. As just one example, most internet service providers aren't built to achieve five nines of availability. Therefore, multiple service providers (with no common point of failure) are required for the application to be 99.999% available to a specific end customer.

In addition, the application and all of its dependencies needs to be built and tested to this level of availability, also avoiding single points of failure. This will require extensive custom development, because many software libraries and systems are not built to five nines availability. The whole system will require exhaustive testing for failure triggers. Because 99.999% availability provides for less than 5 minutes of downtime per year, every operation performed to the system in production will need to be automated and tested with the same level of care. With a 5 minute per year budget, human judgment and action is completely off the table for failure recovery. The system must automatically recover under every situation. Therefore, the production environment will by necessity be slow-moving, with each change tested in a full-scale replica pre-production environment (itself adding significant cost.)

Applications that truly require 99.999% availability can be built on AWS, as the following example illustrates.

99.999% available application. Let's create an ATM network to dispense cash to customers. It consists of custom external devices (ATMs), connected via a network to the host processor operated by the merchant bank that owns the ATM. The merchant bank maintains a cash account for the balance of the machine called a host processor account. The host processor has redundant connectivity to all the banks and banking networks that are to be provided.

The devices themselves are not available all of the time, nor is the network connectivity of any single device, so you deploy a large number of them to enable a customer to easily use a different device if one is down or lacks connectivity. In "availability-speak", they are redundant and fail independently.

The host processor is actually at least two computers and storage that are deployed across independent AWS Regions, with synchronous replication between the Regions. The host processors have redundant connections to the merchant bank and banking networks, and the host processors have standby copies in an independent location. When cash is required, the host processor requests an Electronic Funds Transfer to take the money from the customer's account and put it in the host processor account. After it has the funds transferred, it will send a signal to the ATM to dispense the money. It then initiates an Automated Clearing House (ACH) transfer of those funds to the merchant bank's account over the redundant connectivity to the merchant bank. This will reimburse the merchant bank for the funds dispensed. If a problem happens to the connectivity between the ATM and the host processor or the ATM machine itself after the funds have been transferred to the host processor, it can't tell the ATM to dispense the cash. It will transfer the funds back to the customer's account. The ATM will timeout the transaction and then mark itself out of service.

Multiple AWS Direct Connect connections will be established from the host processors on AWS to the merchant bank. The connectivity from the ATM machines will also be run through redundant Direct Connect providers to the host processor. If the redundant connectivity to the merchant back is severed from both host processors, they will durably store the request for the ACH transfer until the connection is restored, while marking the ATMs that it operates out of service.

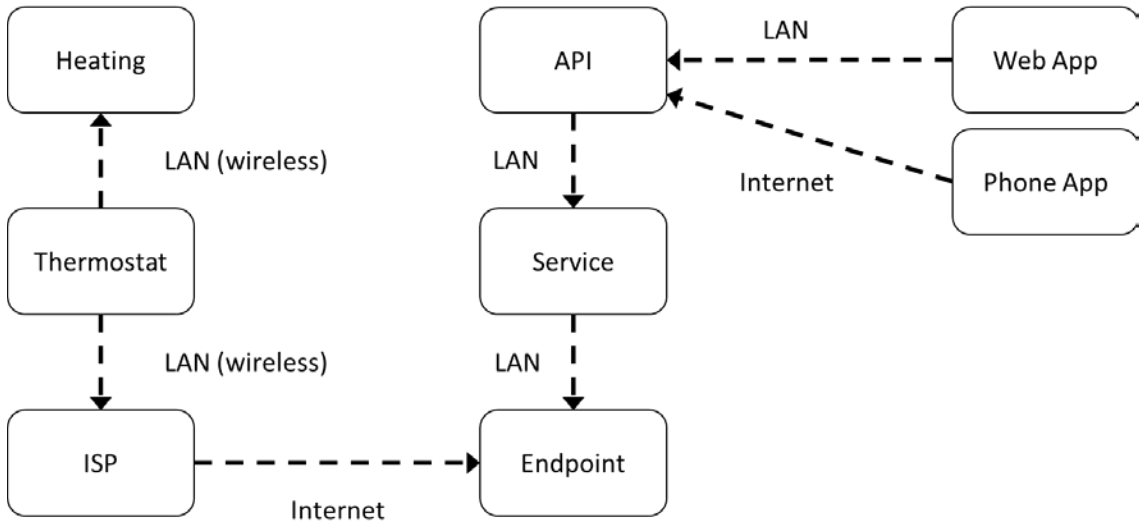
This application can be built and operated on AWS. However as discussed earlier, the costs will be considerable. For most applications, we recommend starting by posing a few simple questions:

- What problems are you trying to solve?
- What specific aspects of the application require specific levels of availability?

- What amount of cumulative downtime can this workload realistically accumulate in a year?
- In essence, what's the real impact of the system being unavailable?

Let's explore an example where you might initially assume the application needs to be 99.999% available, but in reality it can be successful despite a lower availability design goal.

Let's create a smart home heating product. It consists of a mobile application, and a wireless thermostat that is electrically connected to the heating system. The thermostat has a connection to your control endpoint on the internet. Your app uses your API on the internet to send actions to the thermostat. Of course, your users will expect that turning the heating on will always work. They need five nines of availability. How might we deliver that availability? Consider the required architecture for that level of availability:



What if their internet service provider (ISP) has an interruption? To really be available to your customers, you would need a redundant internet connection over mobile. This increases the cost of your thermostat, production costs, running costs, and the complexity of the code that runs on it. You will also have to test that this redundancy switches correctly. And then you need to look at other points of failure in this design. What happens when you need to update the operating system that the "Service" runs on? Or if you need to deploy a new version? What if you need to reconfigure your datacenter network? Or if you need to add more storage? Alternatively, you could have a physical override button on the thermostat for when the internet connection is down.

This is an example where expressing a reliability requirement without considering scope and costs and calculating your return on investment (ROI) could lead you down the wrong path. For example, the thermostat needs five-nines of availability, not the whole architecture. In your analysis, you should be asking questions about unspoken assumptions. Do you have customers at all hours that will not come back to conduct business at another time if you have an interruption? Could you use a lower level of availability with a fallback mechanism to handle failures?

In most applications, there are numerous potential sources of interruption that need to be considered. At higher levels of availability, the detection and response to these interruptions must be fully automated.

The following table list common sources of interruption:

Category	Description
Hardware failure	Failure of any hardware component in the system, including in hosts, storage, network, or elsewhere.
Deployment failure	Failure caused directly as a result of a software, hardware, network, or configuration deployment. This includes both automated and manual changes. The rest of the buckets specifically do not meet this definition.
Load induced	Load related failures can be triggered by a change in behavior, either of a specific caller or in the aggregate, or by the service reaching a tipping point. Load failures can occur in the network.
Data induced	An input or entry is accepted by the system that it can't process ("poison pill")
Credential expiration	Failure caused by the expiration of a certificate or credential.
Dependency	Failure of a dependent service results in failure of the monitored service.
Infrastructure	Power supply or environmental condition failure has an impact on hardware availability.
Identifier exhaustion	Exceeding available capacity, a throttling limit was hit, an ID ran out, or a resource that is vended to customers is no longer available

Achieving 99.999% availability means mastering all of the sources of interruption listed here and automating all human intervention out of

operational processes. It means testing literally every aspect of your application including anticipating ways that your customers will use it that most people could hardly dream of. It means deploying and maintaining canaries that constantly test your application, and frequently doing automated fail-over testing to ensure that each part of your network performs properly under these conditions. It means both unit-level and workflow/transaction monitoring of both success and failure, and it means alarming and log analysis, auto-rollback, and automatic system recovery capabilities that include every dependent service, network connection, and piece of infrastructure between you and your customers.

Upon deep analysis, the work involved in achieving and maintaining high availability applications seems daunting. That often leads to a more refined definition and prioritization of requirements:

- What are the most valuable transactions to your customers and to your business?
- Can you predict when certain geographies have the greatest demand?
- What times of day, week, month, quarter, or year are your peak times?

The good news is that AWS provides numerous services that abstract the work required to do many of these things, provided the right services are chosen to meet your availability design goals. Defining the right design goal for availability will start you down the path of how to implement your application and identify what the additional costs are to achieve that availability. The remainder of this whitepaper will help you think about selecting the right design goal, and then choosing AWS services, building software, and constructing operational practices that meet your goals.

The remainder of this section is presented in four parts:

- Understanding Availability Needs
- Application Design for Availability
- Operational Considerations for Availability
- Example Implementations for Availability Goals

We'll explore how availability needs influence your architecture in "*Understanding availability needs*." In "*Application design for availability*" we look at common techniques we apply to improve availability. We talk about methods of updating your application that can minimize availability impacts in "*Operational Considerations for Availability*." Finally, in "*Example Implementations for Availability Goals*" we illustrate how using different methods can improve your availability.

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 so is to focus 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 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 meta-data 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.

Many of our customers take a similar approach to critically evaluating their applications and identifying sub-components with different availability needs. With this information in hand, availability design goals are then tailored to the sub-component, and work is done to meet the specific design goal of each sub-component. Naturally, components that have higher availability design goals will necessitate deeper investment in the engineering, testing, and operations automation.

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 is 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.

Application Design for Availability

In the years that we've operated Amazon.com and AWS, we've gathered deep experience in designing applications for availability. While there are many lessons to be learned, the five most common practices we apply to improve availability are following:

- Fault Isolation Zones
- Redundant components
- Micro-service architecture
- Recovery Oriented Computing
- Distributed systems best practices

The following sections dive deep on each practice.

Fault Isolation Zones

As described above, one of the most well-known and widely used techniques for increasing a system's availability beyond the availability of individual components is to make use of multiple independent components in parallel. (A common example is the use of multiple AWS Availability Zones.) When building a system that relies on redundant components, it's important to ensure the components operate independently, and in the case of AWS Regions, autonomously. Theoretical availability calculations are only valid if this holds true.

AWS has multiple constructs that provide different levels of independent, redundant components. Starting at the lowest levels, to strengthen data plane availability, AWS partitions resources and requests via some dimension, such as a resource ID. These partitions (which we refer to as “cells” but others may call “shards” or “stripes”) are designed to be independent and further contain faults to within a single cell. To do so, it’s 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 reduce the independence of cells (and thus the assumed availability improvements of doing so). As one example, Amazon Route53 uses the concept of shuffle sharding to isolate customer requests into cells.

AWS also employs the fault isolation construct of Availability Zones (AZs). AWS Regions are composed of two or more Availability Zones that are designed to be independent. Each Availability Zone is separated by a large physical distance from other zones to avoid correlated failure scenarios due to environmental hazards like fires, floods, and tornadoes. Each Availability Zone 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. Despite being geographically separated, Availability Zones are located in the same regional area. This enables synchronous data replication (for example, between databases) without undue impact on application latency. This allows customers to use Availability Zones in an active/active or active/standby configuration. Availability Zones are independent, and therefore application availability is increased when multiple AZs are used. Some AWS services (including the EC2 instance data plane) are deployed as strictly zonal services where they have shared fate with the Availability Zone as a whole. These services are used to independently operate resources (instances, databases, and other infrastructure) within the specific Availability Zone. AWS has long offered multiple Availability Zones in our Regions.

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.

Recommendation
When your application relies on the availability of control plane APIs during a disruption of one **Availability Zone**, use API filters to request results for a single Availability Zone with each API request (for example, with DescribeInstances.)

The most pervasive fault isolation construct is that of the AWS Region. Regions are designed to be autonomous, with dedicated copies of services deployed in each Region. Regional AWS services internally use multiple Availability Zones in an active/active configuration to achieve the availability design goals we establish.

While we provide customers capability to operate services cross-Region (for example, cross-Region replication for Amazon Simple Storage Service (Amazon S3) and the ability to copy various snapshots and Amazon Machine Images (AMIs) to other Regions), we do so in ways that preserves the 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 Route53), 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. Appendix A provides a table of design goals for availability of selected services, in both single Availability Zone and Multi-AZ configurations. You can use this information to guide your design goals for your applications.

Redundant Components

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.

Micro-Service Architecture

At AWS, we have built our systems using a concept called micro-services. While micro-services have several attractive qualities, the most important benefit for availability is that micro-services are smaller and simpler. They allow you to differentiate the availability required of different services, and thereby focus investments more specifically to the micro-services that have the greatest availability needs. For example, to deliver product information pages on Amazon.com ("detail pages"), hundreds of micro-services 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 actually not required to provide an experience where a customer can buy a product.

Microservices take the concept of service-oriented architecture to a 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.

Recommendation
Isolate discrete functionality into services with a "contract" (API and performance expectations).

There are effects to consider when deploying a micro-service architecture. One is that you now have a distributed compute architecture that can make it harder to achieve end-user latency requirements and there is additional complexity in debugging and tracing of user interactions. The AWS X-Ray service 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 you are managing.

Recovery-Oriented Computing

Complementing the AWS focus on building fault isolation zones and avoiding single points of failure, we also work to minimize the disruption time when failures do occur. Since impact duration is a primary input to calculating availability, reducing recovery time has a direct impact on improving availability. Recovery-Oriented Computing (ROC) is the term applied to systematic approaches to improving recovery.

ROC identifies the characteristics in systems that enhance recovery. These characteristics are: isolation and redundancy, system wide ability to roll back changes, ability to monitor and determine health, ability to provide diagnostics, automated recovery, modular design, and ability to restart. We have addressed isolation and redundancy and modular design in the previous sections. In the “Operational Considerations for Availability” section, we will talk about the ability to roll back changes, monitoring, and diagnostics. In this section, we discuss monitoring for health, automated recovery and the ability to restart.

ROC acknowledges that many different types of failures occur in systems. 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, ROC suggests focusing on having the right mechanisms to detect failures (such as Elastic Load Balancing or Route53 health checks). After a failure occurs ROC would apply one of a small number of well-tested recovery paths.

In systems that apply a recovery-oriented approach, many different categories of failures are mapped to the same recovery strategy. For example, applying ROC, we would 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”, ROC would apply a similar strategy of retrying with exponential back-off. Another example is the use of Auto Scaling with Amazon Elastic Compute Cloud (Amazon EC2) to manage fleet capacity. An instance may fail due to hardware failure, operating system bug, memory leak, or other causes. Rather than building custom remediation for each, treat any as an instance failure, terminate the instance, and allow Auto Scaling to replace the instance.

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 are incorrect. The capacity of the secondary data store, 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 failover to the standby regularly, regardless of need.

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 independence of Availability Zones. We use automation extensively in our operational procedures. We also optimize our replace-and-restart functionality to recover quickly from interruptions.

Distributed Systems Best Practices

As we apply the approaches we have discussed in these sections, including micro-service architecture and the use of fault isolation zones, we recognize that many systems built today are distributed systems. They rely on communications networks to interconnect components. Particularly when traversing longer distances or intermediary networks, these systems can have high latency or loss. Individual services may see spikes of requests that temporarily overwhelm their ability to respond. There are a number of best practices that can be applied to allow these services to continue to operate normally in the presence of these "normal" issues.

These best-practice patterns include the following:

Throttling: This is a defensive pattern to respond to an unexpected increase in demand, typically on a web service. Some requests will be honored, but the rejected requests will return a message indicating they have been throttled, with the expectation they will 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.

Retry with exponential fallback: This is the invoking side of the throttling pattern we just discussed. AWS SDKs implement this by default, and can be configured. The pattern is to pause and then retry at a later time. If it fails again, pause longer and retry. This increase in pause time is often called "backing off." After a configured number of attempts or elapsed time, it will quit retrying and return failure.

Fail fast: Simply return an error as soon as possible. This will allow releasing of resources associated with requests and can often allow a service to recover if it is running out of resources. It's preferable to fail fast rather than allowing requests to be queued. Queues can be created at multiple levels of a system, and can seriously impede an ability to quickly recover. Be very mindful of places where queues exist (they often hide in workflows or in work that's recorded to a database).

Use of idempotency tokens: In a distributed system, it's easy to perform an action at most once, or at least once. But it's hard to guarantee an action is performed exactly once. A common approach to do so is the use of idempotency tokens in APIs. In doing so, services can receive a mutating request one or more times without creating duplicate records or side effects. Callers issue API requests with an idempotency token; the same token is used whenever the request is repeated (for example, due to a timeout and retry.) When receiving a request that has already been processed, an idempotent API uses the token to determine the work has already been completed, and then returns a response identical to the response that's returned when the work is completed for the first time. It is more resilient to build systems with idempotency than to build systems that assume an action must occur exactly once.

Constant work: Systems can fail when there are rapid changes in load. If you know your service needs to process 100 units of work done per second at peak, then you should design and tune your system for 100 units of work. If there is work to be done, it takes one of the slots. But if not, you put "filler" work in the slot just so you know you're always doing 100 units per second. An example is a video player that plays data in a buffer. If you have no work to perform because no new data has come in, you may be able to process the same data you last received again and effectively render the same video frame, performing the same work.

Circuit breaker: In certain situations, a service has a need to make remote requests on a best effort basis, but does not want to take a hard dependency, which would include the dependency's availability in the computation of the invoking service's availability design goal. In these cases, one solution is to use a monitoring loop and circuit breaker for each remote request. 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 to avoid further latency impact or availability impact. When open, the dependency is ignored or results are replaced with locally-available data (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.

Bi-modal behavior and static stability: Distributed systems can be impacted by negative feedback loops that are triggered by one failure. For example, a network timeout could cause a system to attempt to refresh the configuration state of the entire system. This would add unexpected load to another component, which might then cause it to fail, triggering other unexpected consequences. We refer to this as "bi-modal" behavior, because the system has different behavior under normal and failure modes. To counteract, this behavior, we prefer building systems that are statically stable and operate in only one mode. They maintain enough internal state to continue operating as they were before the failure without adding additional load to the system. These systems may end up performing less work during certain failures (which is desirable). Another example of this type of system is one that uses Amazon EC2 for instance capacity. Systems often assume that if an instance or Availability Zone fails, they will respond by simply launching new instances. However, this approach means that during failure, the system will be doing much different work from usual. Instead, we recommend using Elastic Load Balancing or Amazon Route53 health checks to shift load away from failed instances, and use Auto Scaling to asynchronously replace them.

Operational Considerations for Availability

Experience and data from many IT workloads highlights the importance of operations and human processes on application availability. Despite all of the investments in software and hardware, an erroneous configuration or misstep in a process can frequently undo these efforts. When designing software to meet availability design goals, it’s important to plan the automated or human processes used in the full lifecycle of the application. This includes deployment of new versions, operation of the service, refreshing the underlying infrastructure, and replacing failed infrastructure.

Testing is an important part of the delivery pipeline. Aside from common unit tests and functional tests that are performed at component levels, it is important to perform sustained load testing. Load tests should discover the breaking point of your workload, test performance, and perform fault injection testing. In addition, your monitoring service must be able to add or remove monitoring of capabilities that are added or deprecated. 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 applications performance to its SLAs and provide data in the event of an interruption or other operational anomaly.

Automate Deployments to Eliminate 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 our 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.

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.

These are deployment patterns that minimize risk:

- Canary deployment
- Blue-Green deployment
- Feature toggles
- Failure isolation zone deployments

Canary deployment is the practice of directing a small number of your customers to the new version and scrutinizing deeply any behavior changes or errors that are generated. You can remove traffic from the canary if you have critical problems and send the users to the previous version. If the deployment is successful, you can continue to deploy at a desired velocity, while monitoring for the same changes and errors, until you are fully deployed. AWS Code Deploy can be configured with a deployment configuration that will enable a canary deployment.

Blue-Green deployments are 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 fail back to the old version if you see problems with the deployment. You can also use fractions of your traffic to each version to dial up the adoption of the new version. AWS Code Deploy can be configured with a deployment configuration that will enable a blue-green deployment.

Feature toggles are configuration options on an application. You can deploy the software with a feature turned off, so that customers don't see the feature. You can then turn the feature on, 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.

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.

Testing

The testing effort should be commensurate with your availability goals. Your application's resiliency to transient failures of dependencies should be tested for durations that may last from less than a second to hours. Testing to ensure that you can meet your availability goals is the only way you can have confidence that you will meet those goals. Our experience is that canary testing that can run constantly and simulate customer behavior is among the most important testing processes. You should unit test, load test, performance test, and simulate your failure modes while under these tests. Don't forget to test for external dependency unavailability, and deployment failures. Achieving very high availability requires implementing fault tolerant software patterns, and testing that they are effective.

Other modes of degradation may cause reduced functionality and slow responses, often resulting in a brown out of your services. Common sources of this degradation are increased latency on critical services and unreliable network communication (dropped packets). You might want to use the ability to inject random failures into your system, including component failures, networking effects such as latency and dropped messages, and DNS failures such as being unable to resolve a name or not being able to establish connections to dependent services.

Netflix has provided some example open source software that can be a basis for this type of testing. You can use their software or develop your own for simulating failure modes. For simulating conditions that might produce brownouts, you can use extensions to common proxies to introduce latency, dropped messages, etc., or you can create your own.

Monitoring and Alarming

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 customer knows 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 also look for outlying data points because this can be another indication of impending problems. This is commonly known as percentile monitoring. If your average is acceptable, but one in 100 of your requests causes extreme latency, when your traffic grows it will eventually become a problem.

In addition, monitor all of your external endpoints from remote locations to ensure that they are independent of your base implementation. We have seen improvement in time to detection of problems with use of “user canary” applications, which execute some number of common tasks performed by consumers of the application. They can be implemented in both graphic user interfaces and web services. They all must complete within a very short time, with a target of 1 second. These must be carefully selected so that they don’t overload the application during testing. The reason to have only short duration tasks is so you can run them once per minute, which enables you to detect a problem before it is visible to users.

While monitoring from within an operating system is well understood, monitoring in the cloud offers new opportunities. Instead of using old de-facto standard methods like SNMP, cloud providers have developed customizable hooks and insights into everything from instance performance to network layers, down to request APIs themselves.

Monitoring at AWS consists of five distinct phases:

1. Generation
2. Aggregation
3. Real-time processing and alarming
4. Storage
5. Analytics

Generation

First, determine which services and/or applications require monitoring, define important metrics and how to extract them from log entries if necessary, and finally create thresholds and corresponding alarm events. 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, Classic Load Balancers, Application Load Balancers, 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 on any or all elastic network interfaces (ENIs) within a VPC.
- AWS CloudTrail logs all API events on an account-by-account basis.
- Amazon CloudWatch Events 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.
- Amazon Machine Learning (Amazon ML), 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.

Aggregation

Amazon CloudWatch and Amazon S3 serve as the primary aggregation and storage layers. For some services, like Auto Scaling and ELB, 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 or AWS CloudTrail, event data is forwarded to CloudWatch Logs and you need to define and apply filters to extract metrics from the event data. This gives you time series data, and you can define an array of CloudWatch alarms to trigger alerts.

Real-Time Processing and Alarming

Alerts can trigger Auto Scaling events, so that clusters react to changes in demand. 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. Alerts can be sent to Amazon Simple Queue Service (Amazon SQS), which can serve as an integration point for third-party ticket systems. Finally, AWS Lambda can also subscribe to alerts, providing users an asynchronous serverless model that reacts to change dynamically.

Storage and Analytics

Amazon CloudWatch Logs also supports subscriptions that allow data to flow seamlessly to Amazon S3. As CloudWatch logs and other access logs arrive in Amazon S3, you should consider using Amazon EMR to gain further insight and value from the data itself. If your data is written in a supported manner, Amazon S3 Select or Amazon Athena can be used to query the data. Amazon S3 Select supports Comma-Separated Values (CSV) or JavaScript Object Notation (JSON) documents with or without GZIP compression. Amazon Athena supports a large array of formats. For more information, see Supported SerDes and Data Formats in the Amazon Athena User Guide.

There are a number of tools provided by partners and third parties that allow for aggregation, processing, storage, and analytics. Some of these tools are New Relic, Splunk, Loggly, Logstash, CloudHealth, and Nagios. However, generation outside 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 retention requirements for monitoring data, and then apply lifecycle polices 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 Glacier for long-term storage, and then expiration, after the end of the retention period is reached.

Key AWS Services

The key AWS service that supports monitoring is **Amazon CloudWatch**, which allows for easy creation of alarms that trigger scaling actions. In addition, **AWS X-Ray** can be integrated with your applications to provide visibility into the distributed interaction of requests with your applications.

The following services and features are also important:

- **Amazon S3**: Acts as the storage layer, and allows for lifecycle policies and data management.
- **Amazon EMR**: Use this service to gain further insight into log and metric data.
- **Amazon Athena**: Use this service to gain further insight into data that is in support formats.

Operational Readiness Reviews

Operational Readiness Reviews (ORRs) are an important exercise to confirm applications are ready for production operations. Teams often start with an ORR checklist during early stages of application development. This enables them to keep in mind the requirements of their operational environment prior to asking for a production deployment. 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. An ORRs for one application should incorporate lessons learned and best practices from other applications.

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

Auditing

Auditing your monitoring will ensure that you know when an application is meeting its availability goals. Root Cause Analyses require 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.

- **AWS Config:** You can see what AWS infrastructure were used at 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. Establishing a regular cadence for operational performance reviews and knowledge sharing will enhance your ability to achieve higher performance from your operational teams.

Example Implementations for Availability Goals

In this section, we'll review system 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 will provide an example implementation. These can be deployed using containers or virtual machines, but the approaches are the same. In this section, we will address the remaining topics of the reliability pillar. Specifically, in each scenario, we will demonstrate how to:

- Adapt to changes in demand
- Use monitoring
- Deploy changes
- Back up data
- Implement resiliency
- Test resiliency
- Recover from disaster

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 can only adopt services with higher availability themselves. See the Appendix for the design goals for the respective services.

Single Region Scenarios

2 9s (99%) Scenario

We will start our availability and reliability examples with applications that are helpful to the business, but it is only an *inconvenience* if the applications are unavailable. This type of application can vary from internal tooling systems, internal knowledge management systems, and project tracking systems, to actual customer-facing features that are served from an experimental service, with a feature toggle that can hide the service if needed.

These applications can be deployed with one Region and one Availability Zone. We will deploy the software, including the database, to a single instance. 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 and ensuring the ability to use them on a regular basis using a runbook. We will configure versioning on our Amazon S3 objects and remove permissions for deletion of the backups. We will use an Amazon S3 bucket lifecycle policy to archive or permanently delete according to our requirements. We will use AWS CloudFormation to define our infrastructure as code, and specifically to speed up reconstruction in the event of a failure. During failures we will 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 can be 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. The deployment pipeline of new software is scheduled, with some unit testing, but mostly white-box/black-box testing of the assembled system. Software updates will be manually performed using a runbook, with downtime required for the installation and re-start of the service. If a problem happens during deployment, the runbook describes how to roll back to the previous version. We will have playbooks for common hardware failures, urgent software updates, and other disruptive changes. 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. The correction of the error will be done using analysis by the operations and development teams, and the correction of the error will be deployed when the fix is prioritized and completed.

Let's see what the implications on availability of recovery time are. 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 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 the upper limit on availability is 99.9%. The actual availability will also depend on the real rate of failure, duration of failure and how quickly each factor 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%.

Here is how we addressed the remaining reliability pillar topics:

Topic	**Implementation**
Adapting to changes in demand	Vertical scaling via re-deployment.
Monitoring	Site health check only; no alerting.
Deploying changes	Runbook for deploy and rollback.
Backups	Runbook for taking and restoring.
Implementing resiliency	Complete rebuild; restore to backup.
Testing resiliency	Complete rebuild; restore to backup.
Disaster recovery	Encrypted backups, restore to different Availability Zone if needed.

3 9s (99.9%) Scenario

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

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. 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.

The infrastructure deployment technologies remain the same. 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 software/application 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. Software updates will be automated, not using canary or blue/green deployment patterns, but rather, using the replace in place. The decision to rollback will be made using the runbook.

Delivery of new software is on a fixed schedule of every two to four weeks. Monitoring will be expanded to alert on the availability of the web site 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.

Runbooks exist for total system recovery and common reporting. We will have playbooks for common database problems, security-related incidents, failed deployments, and 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 when the fix is developed.

Let’s see what the implications on availability of recovery time are. We assume that at least some failures will require a manual decision to execute recovery. However with greater automation in this scenario we assume only two events per year will require this decision. We take 30 minutes to decide to execute recovery, and assume 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 the upper limit on availability is 99.95%. The actual availability will also depend on the real rate of failure, duration of failure and how quickly each factor 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%.

Here is how we addressed reliability pillar topics:

Topic	**Implementation**
Adapting to changes in demand	ELB for web and auto scaling application tier; resizing Multi-AZ RDS.
Monitoring	Site health check only; alerts sent when down.
Deploying changes	Automated deploy in place and runbook for rollback.
Backups	Automated backups via RDS to meet RPO and runbook for restoring.
Implementing resiliency	Auto scaling to provide self-healing web and application tier; RDS is Multi-AZ.
Testing resiliency	ELB and application are self-healing; RDS is Multi-AZ; no explicit testing.
Disaster recovery	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 to be able to absorb failures without needing to procure additional resources. This availability goal is for mission critical applications that are main or significant revenue drivers for a corporation, such as an e-commerce 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 Route53 DNS. We should not need to create any new infrastructure, whether it is creating/modifying an S3 bucket, creating new IAM policies (or modifications of policies), or modifying Amazon ECS task configurations.

We recommend three Availability Zones for this approach. Using a three Availability Zone deployment, each Availability Zone has static capacity of 50% of peak. Two AZs could be used, but the cost of the statically stable capacity would be more because both Availability 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.

The application will be built using the software/application resiliency patterns in all layers. For these applications, engineering for read availability over write availability of primary content is also a key architecture decision. The application is also implemented in deployment fault isolation zones. 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 into each isolation zone singularly. The deployments are fully automated, including a roll back if KPIs indicate a problem. 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.

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 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. That team will identify the correction of error of any unexpected failure and prioritize the fix to be deployed after it is implemented. We will also engage with AWS Support for Infrastructure Event Management offering.

Let's see what the implications on availability of recovery time are. We assume that at least some failures will require a manual decision to execute recovery, however with greater automation in this scenario we assume 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 recovery is completed within five minutes. This implies 15 minutes to recover from failure. Assuming two per year, our estimated impact time for the year is 30 minutes.

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

Here is how we addressed reliability pillar topics:

Topic	**Implementation**
Adapting to changes in demand	ELB for web and auto scaling application tier; resizing Multi-AZ RDS.
Monitoring	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.
Deploying changes	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.
Backups	Automated backups via RDS to meet RPO and automated restoration that is practiced regularly in a game day.
Implementing resiliency	Implemented fault isolation zones for the application; auto scaling to provide self-healing web and application tier; RDS is Multi-AZ.
Testing 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.

Disaster recovery	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 independence. It should be a very thoughtful decision to pursue this path. That said, regions provide a very 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.

3 ½ 9s (99.95%) with a Recovery Time between 1 and 30 Minutes

This availability goal for applications requires extremely short downtime and very little data loss for 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 "Hot Standby" approach across two AWS Regions. We will deploy the workload to both Regions, with our passive site scaled (and kept eventually consistent) to receive same traffic load as our active site. Both Regions will be *statically stable*. The applications should be built using the software/application resiliency patterns. We'll need to create a lightweight *routing* component that monitors both our application health and any regional hard dependencies we have. This component will also handle automation of failover, and stop replication from the former active Region. During failover, we will route requests to a static website using DNS failover until recovery in the second Region. The failover will use a health check of the web site over all by checking for an HTTP 200 OK status on the home page. Software updates will be automated using canary or blue/green deployment patterns.

Delivery of new software is on a fixed schedule of every two to four weeks. In addition, 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. 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 validate the architecture through game days using runbooks. We will also engage with AWS Support for Infrastructure Event Management.

Let's see what the implications on availability of recovery time are. We assume that at least some failures will require a manual decision to execute recovery, however with good automation in this scenario we assume only 2 events per year will require this decision. We take 20 minutes to decide to execute recovery, and assume recovery is completed within 10 minutes. This implies 30 minutes to recover from failure. Assuming 2 per year, our estimated impact time for the year is 60 minutes.

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

Here is how we addressed reliability pillar topics:

Topic	**Implementation**
Adapting to changes in demand	ELB for web and auto scaling application tier; resizing Multi-AZ RDS; this is synchronized between AWS Regions for static stability.
Monitoring	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.
Deploying changes	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.
Backups	Automated backups in each AWS Region via RDS to meet RPO and automated restoration that is practiced regularly in a game day.
Implementing resiliency	Auto 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.

Testing 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.
Disaster recovery	Encrypted backups via RDS, with replication between two AWS Regions. 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

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 almost 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 is possible to have no data loss. This might require you to add application logic and latency to ensure you have consistent data between geographic locations, as well as the capability to move or copy data between partitions. Performing this partitioning may be easier if you use a NoSQL database.

We can improve availability further by using an “Active/Active” or “Multi-master” approach across multiple AWS Regions. The workload will be deployed in all desired Regions that are *statically stable* with the *routing* layer directing traffic to geographic locations that are healthy and automatically changing the destination when a location is unhealthy, as well as temporarily stopping the data replication layers. Amazon Route53 offers 10 second interval health checks and also offers TTL on your record sets as low as one second.

The applications should be built using the software/application resiliency patterns. It is 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.

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 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. Data stores must be replicated between the Regions in a manner which 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. We will validate the architecture 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. That team will identify the correction of error of any unexpected failure and prioritize the fix to be deployed after it is implemented. We will also engage with AWS Support for Infrastructure Event Management.

Let’s see what the implications on availability of recovery time are. 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%.

Here is how we addressed reliability pillar topics:

Topic	**Implementation**
Adapting to changes in demand	ELB for web and auto scaling application tier; resizing Multi-AZ RDS; this is synchronized between AWS Regions for static stability.
Monitoring	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.
Deploying changes	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.
Backups	Automated backups in each AWS Region via RDS to meet RPO and automated restoration that is practiced regularly in a game day.
Implementing 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.

Testing 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.
Disaster recovery	Encrypted backups via RDS, with replication between two AWS Regions. Restoration is to the current active AWS Region, is practiced in a game day, and is coordinated with AWS.

Conclusion

Whether you are new to the topics of availability and reliability or a seasoned veteran seeking insights to maximize your mission critical service's availability, we hope this section 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. 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 via our website.

Resources

Refer to the following resources to learn more about AWS best practices in this area.

Documentation:

- DynamoDB: Global Tables
- DynamoDB: On-Demand Backup and Restore
- DynamoDB: Point-in-Time Recovery
- RDS: Replicating a Read Replica Across Regions
- S3: Cross-Region Replication
- Route 53: Configuring DNS Failover
- Amazon EBS Snapshot Copies
- AMI Copies
- Amazon RDS: Cross-region backup copy
- Using AWS for Disaster Recovery
- AWS Architecture Center
- AWS X-Ray Documentation
- Using API Gateway to Throttle Requests
- Working with Deployment Groups (CodeDeploy)
- Blue/Green Deployments on AWS
- Canary Blue/Green Deployment on ECS
- Blue/Green Deployment on ECS
- Shuffle Sharding: Massive and Magical Fault Isolation
- Add Scaling to Services You Build on AWS

Books and External Links:

- Michael Nygard “Release It! Design and Deploy Production-Ready Software”
- Robert S. Hammer “Patterns for Fault Tolerant Software”
- Andrew Tanenbaum and Marten van Steen “Distributed Systems: Principles and Paradigms”

- Adaptive Queuing Pattern: Fail at Scale
- Blue Green Deployment
- Canary Release
- Feature Toggles
- Microservice Trade-Offs
- Recovery Oriented Computing
- Calculating Total System Availability
- Netflix Simian Army
- Percentile Monitoring (An example on latency monitoring)

Contributors

The following individuals and organizations contributed to this document:

- Philip Fitzsimons, Sr Manager Well-Architected, Amazon Web Services
- Rodney Lester, Reliability Lead, Well Architected Amazon Web Services
- Michael Wallman, Sr. Professional Services Consultant, Amazon Web Services
- Kevin Miller, Director Software Development, Amazon Web Services
- Shannon Richards, Sr. Technical Program Manager, Amazon Web Services

Document Revisions

Date	Description
June 2018	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.
March 2018	Changed DynamoDB Cross Region solution to DynamoDB Global Tables Added service design goals
December 2017	Minor correction to availability calculation to include application availability

Date	Description
November 2017	Updated to provide guidance on high availability designs, including concepts, best-practices and example implementations.
November 2016	First publication

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 Route53, and the Identity & 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 provide independence between Availability Zones (AZs); in these cases we provide the availability design goal for a single AZ, and when any two (or more) AZs are used.

NOTE: The numbers in the table below 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%
AWS CloudFormation	Service	99.950%
Amazon CloudFront	Control Plane	99.900%
	Data Plane (content delivery)	99.990%

Service	Component	Availability Design Goal
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%
AWS Data Pipeline	Service	99.990%
Amazon DynamoDB	Service (standard)	99.990%
	Service (Global Tables)	99.999%
Amazon EC2	Control Plane	99.950%
	Single AZ Data Plane	99.950%
	Multi AZ Data Plane	99.990%
Amazon ElastiCache	Service	99.990%
Amazon Elastic Block Store	Control Plane	99.950%
	Data Plane (volume availability)	99.999%
Amazon Elasticsearch	Control Plane	99.950%
	Data Plane	99.950%
Amazon EMR	Control Plane	99.950%
Amazon Glacier	Service	99.900%
AWS Glue	Service	99.990%
Amazon Kinesis Streams	Service	99.990%
Amazon RDS	Control Plane	99.950%
	Single AZ Data Plane	99.950%
	Multi AZ Data Plane	99.990%
Amazon Redshift	Control Plane	99.950%
	Data Plane	99.950%
Amazon Route53	Control Plane	99.950%
	Data Plane (query resolution)	100.000%
Amazon S3	Service (Standard)	99.990%
AWS Auto Scaling	Control Plane	99.900%
	Data Plane	99.990%
AWS Batch	Control Plane	99.900%

Service	Component	Availability Design Goal
	Data Plane	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 Direct Connect	Control Plane	99.900%
	Single Location Data Plane	99.900%
	Multi Location Data Plane	99.990%
AWS Elastic File Store	Control Plane	99.950%
	Data Plane	99.990%
AWS Identity & Access Management	Control Plane	99.900%
	Data Plane (authentication)	99.995%
AWS Lambda	Function Invocation	99.950%
AWS Storage Gateway	Control Plane	99.950%
	Data Plane	99.950%
AWS X-Ray	Control Plane (console)	99.900%
	Data Plane	99.950%
EC2 Container Service	Control Plane	99.900%
	EC2 Container Registry	99.990%
	EC2 Container Service	99.990%
Elastic Load Balancing	Control Plane	99.950%
	Data Plane	99.990%
Key Management System (KMS)	Control Plane	99.990%
	Data Plane	99.995%