Reliability Pillar

AWS Well-Architected Framework

Reliability Pillar: AWS Well-Architected Framework

Table of Contents

Reliability Pillar - AWS Well-Architected Framework

Publication date: **October 20, 2022** (*Document revisions* (p. 134))

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

Introduction

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

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

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

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

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

Reliability

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

Topics

Design principles

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

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

Definitions

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

- Foundations
- Workload Architecture

- Change Management
- Failure Management

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

Topics

Resiliency, and the components of reliability

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

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

The other factors impacting workload reliability are:

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

Resiliency is the primary focus of this whitepaper.

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

Availability

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

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

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

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

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

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

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

Availability	Maximum Unavailability (per year)	Application Categories
99% (p. 119)	3 days 15 hours	Batch processing, data extraction, transfer, and load jobs
99.9% (p. 120)	8 hours 45 minutes	Internal tools like knowledge management, project tracking
99.95% (p. 124)	4 hours 22 minutes	Online commerce, point of sale
99.99% (p. 122)	52 minutes	Video delivery, broadcast **workloads**
99.999% (p. 127)	5 minutes	ATM transactions, telecommunications **workloads**

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

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

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

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

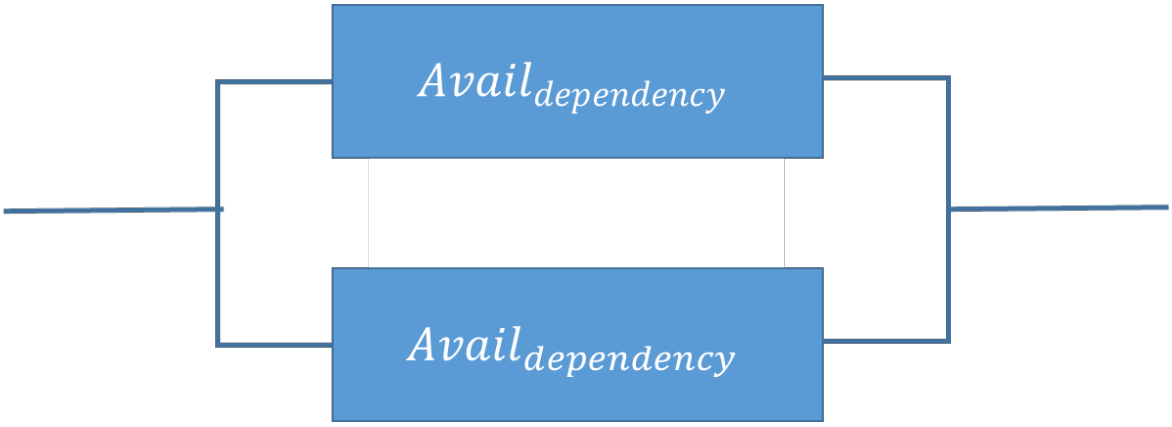
invoking system. This is opposed to a soft dependency, where a failure of the dependent system is compensated for in the application. Where such hard dependencies occur, the invoking system's availability is the product of the dependent systems' availabilities. For example, if you have a system designed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

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

99.99% × 99.99% × 99.99% = 99.97%

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

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



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

99.9999% = 100% − (0.1%×0.1%)

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

Calculating dependency availability. Some dependencies provide guidance on their availability, including availability design goals for many AWS services (see Appendix A: Designed-For Availability for Select AWS Services (p. 137)). But in cases where this isn't available (for example, a component where the manufacturer does not publish availability information), one way to estimate is to determine the **Mean Time Between Failure (MTBF)** and **Mean Time to Recover (MTTR)**. An availability estimate can be established by:

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

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

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

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

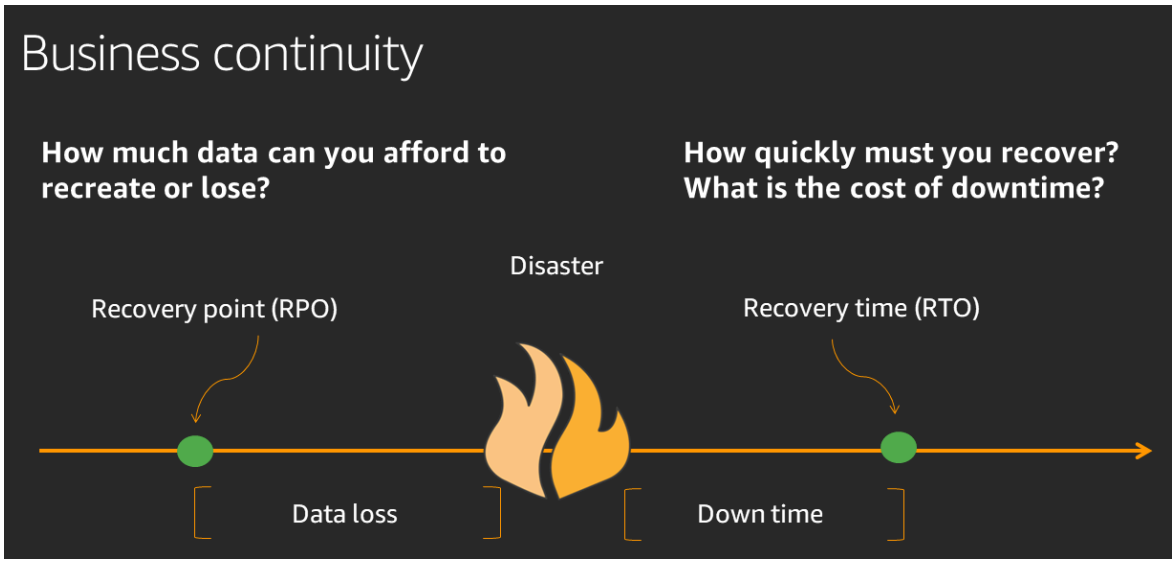
Another way that costs escalate in systems that operate with higher availability design goals is in the selection of dependencies. At these higher goals, the set of software or services that can be chosen as dependencies diminishes based on which of these services have had the deep investments we previously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

Disaster Recovery (DR) objectives

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

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

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



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

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

Understanding availability needs

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

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

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

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

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

Foundations

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

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

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

Topics

Manage service quotas and constraints

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

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

Best practices

REL01-BP01 Aware of service quotas and constraints

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

Service Quotas is an AWS service that helps you manage your quotas for over 100 AWS services from one location. Along with looking up the quota values, you can also request and track quota increases

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

Common anti-patterns:

- Deploying a workload with no regard of the service quotas on the AWS services used.
- Designing a workload without investigating and accommodating for AWS services' design constraints.
- Deploying a workload with significant use that replaces a known existing workload without configuring the necessary quotas or contacting AWS Support in advance.
- Planning an event to drive traffic to your workload, but not configuring the necessary quotas or contacting AWS Support in advance.

Benefits of establishing this best practice: Being aware of the service quotas, API throttling limits, and design constraints will allow you to account for these in your design, implementation, and operation of the workload.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Review AWS service quotas in the published documentation and Service Quotas
  - AWS Service Quotas (formerly referred to as limits)
- Determine all the services your workload requires by looking at the deployment code.
- Use AWS Config to find all AWS resources used in your AWS accounts.
  - AWS Config Supported AWS Resource Types and Resource Relationships
- You can also use your AWS CloudFormation to determine your AWS resources used. Look at the resources that were created either in the AWS Management Console or via the list-stack-resources CLI command. You can also see resources configured to be deployed in the template itself.
  - Viewing AWS CloudFormation Stack Data and Resources on the AWS Management Console
  - AWS CLI for CloudFormation: list-stack-resources
- Determine the service quotas that apply. Use the programmatically accessible information via Trusted Advisor and Service Quotas.

Resources

Related documents:

- AWS Marketplace: CMDB products that help track limits
- AWS Service Quotas (formerly referred to as service limits)
- AWS Trusted Advisor Best Practice Checks (see the Service Limits section)
- AWS limit monitor on AWS answers
- Amazon EC2 Service Limits
- What is Service Quotas?

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas

REL01-BP02 Manage service quotas across accounts and regions

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

Service quotas are tracked per account. Unless otherwise noted, each quota is AWS Region-specific. In addition to the production environments, also manage quotas in all applicable non-production environments, so that testing and development are not hindered.

Common anti-patterns:

- Allowing resource utilization in one isolation zone to grow with no mechanism to maintain capacity in the other ones.
- Manually setting all quotas independently in isolation zones.
- Not ensuring Regionally isolated deployments are sized to accommodate the increase in traffic from another Region if a deployment is lost.

Benefits of establishing this best practice: Ensuring that you can handle your current load if an isolation zone is unavailable can help reduce the number of errors that occur during failover, instead of causing a denial of service to your customers.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Select relevant accounts and Regions based on your service requirements, latency, regulatory, and disaster recovery (DR) requirements.
- Identify service quotas across all relevant accounts, Regions, and Availability Zones. The limits are scoped to account and Region.
- What is Service Quotas?

Resources

Related documents:

- AWS Marketplace: CMDB products that help track limits
- AWS Service Quotas (formerly referred to as service limits)
- AWS Trusted Advisor Best Practice Checks (see the Service Limits section)
- AWS limit monitor on AWS answers
- Amazon EC2 Service Limits
- What is Service Quotas?

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas

REL01-BP03 Accommodate fixed service quotas and constraints through architecture

Be aware of unchangeable service quotas and physical resources, and architect to prevent these from impacting reliability.

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

Common anti-patterns:

- Performing benchmarking for too short of time, utilizing the burst limit, but then expecting the service to perform at that capacity for sustained periods.
- Choosing a design that uses one resource of a service per user or customer, unaware that there are design constraints that will cause this design to fail as you scale.

Benefits of establishing this best practice: Tracking fixed quotes in AWS services and constraints in other parts of your workload, such as connectivity constraints, IP address constraints, and constraints in third-party services, allows you to detect when you are trending toward a quota and gives you the ability to address the quota before it's exceeded.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Be aware of fixed service quotas Be aware of fixed service quotas and constraints and architect around these.
  - AWS Service Quotas

Resources

Related documents:

- AWS Marketplace: CMDB products that help track limits
- AWS Service Quotas (formerly referred to as service limits)
- AWS Trusted Advisor Best Practice Checks (see the Service Limits section)
- AWS limit monitor on AWS answers
- Amazon EC2 Service Limits
- What Is Service Quotas?

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas

REL01-BP04 Monitor and manage quotas

Evaluate your potential usage and increase your quotas appropriately, allowing for planned growth in usage.

For supported services, you can manage your quotas by configuring CloudWatch alarms to monitor usage and alert you to approaching quotas. These alarms can be triggered from Service Quotas or from

Trusted Advisor. You can also use metric filters on CloudWatch Logs to search and extract patterns in logs to determine if usage is approaching quota thresholds.

Common anti-patterns:

- Configuring alarms for when Service Quotas are being approached, but having no process on how to respond to an alert.
- Only configuring alarms for services supported by Service Quotas and not monitoring other services.

Benefits of establishing this best practice: Automatic tracking of the AWS service quotas and monitoring your usage against those quotas will allow you to see when you are approaching a quota limit. You can also use this monitoring data to assess when you might lower quotas to save costs.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Monitor and manage your quotas Evaluate your potential usage on AWS, increase your regional service quotas appropriately, and allow planned growth in usage.
  - Capture current resource consumption (for example, buckets, instances). Use service API operations, such as the Amazon EC2 DescribeInstances API, to collect current resource consumption.
  - Capture your current quotas Use AWS Service Quotas, AWS Trusted Advisor, and AWS documentation.
    - Use AWS Service Quotas, an AWS service that helps you manage your quotas for over 100 AWS services from one location.
    - Use Trusted Advisor service limits to determine your current service limits.
    - Use service API operations to determine current service quotas where supported.
    - Keep a record of quota increases that have been requested, and their status After a quota increase has been approved, ensure that you update your records to reflect the change to the quota.

Resources

Related documents:

- AWS Marketplace: CMDB products that help track limits
- AWS Service Quotas (formerly referred to as service limits)
- AWS Trusted Advisor Best Practice Checks for Service Limits
- AWS limit monitor on AWS answers
- Amazon EC2 Service Limits
- What Is Service Quotas?
- Monitor Service Quotas using Amazon CloudWatch alarms

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas

REL01-BP05 Automate quota management

Implement tools to alert you when thresholds are being approached. You can automate quota increase requests by using AWS Service Quotas APIs, you can automate quota increase requests.

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

Common anti-patterns:

- Tracking the quotas and usage in spreadsheets.
- Running reports on usage daily, weekly, or monthly, and then comparing usage to the quotas.

Benefits of establishing this best practice: Automated tracking of the AWS service quotas and monitoring of your usage against that quota allows you to see when you are approaching a quota. You can set up automation to assist you in requesting a quota increase when needed. You might want to consider lowering some quotas when your usage trends in the opposite direction to realize the benefits of lowered risk (in case of compromised credentials) and cost savings.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Set up automated monitoring Implement tools using SDKs to alert you when thresholds are being approached.
  - Use Service Quotas and augment the service with an automated quota monitoring solution, such as AWS Limit Monitor or an offering from AWS Marketplace.
    - What is Service Quotas?
    - Quota Monitor on AWS - AWS Solution
  - Set up triggered responses based on quota thresholds, using Amazon SNS and AWS Service Quotas APIs.
  - Test automation.
    - Configure limit thresholds.
    - Integrate with change events from AWS Config, deployment pipelines, Amazon EventBridge, or third parties.
    - Artificially set low quota thresholds to test responses.
    - Set up triggers to take appropriate action on notifications and contact AWS Support when necessary.
    - Manually trigger change events.
    - Run a game day to test the quota increase change process.

Resources

Related documents:

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

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas

REL01-BP06 Ensure that a sufficient gap exists between the current quotas and the maximum usage to accommodate failover

When a resource fails, it might still be counted against quotas until it's successfully terminated. Ensure that your quotas cover the overlap of all failed resources with replacements before the failed resources are terminated. You should consider an Availability Zone failure when calculating this gap.

Common anti-patterns:

- Setting service quotas based on current needs without accounting for failover scenarios.

Benefits of establishing this best practice: When events potentially impact availability, the cloud allows you to implement strategies to mitigate or recover from these events. Such strategies often include creating additional resources to replace failed ones. Your quota strategy must accommodate these additional resources.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Ensure that there is enough gap between your service quota and your maximum usage to accommodate for a failover.
  - Determine your service quotas, accounting for your deployment patterns, availability requirements, and consumption growth.
  - Request quota increases if necessary. Plan for necessary time for quota increase requests to be fulfilled.
    - Determine your reliability requirements (also known as your number of 9's).
    - Establish your fault scenarios (for example, loss of a component, an Availability Zone, or a Region).
    - Establish your deployment methodology (for example, canary, blue/green, red/black, or rolling).
    - Include an appropriate buffer (for example, 15%) to the current limit.
    - Plan consumption growth (for example, monitor your trends in consumption).

Resources

Related documents:

- AWS Marketplace: CMDB products that help track limits
- AWS Service Quotas (formerly referred to as service limits)
- AWS Trusted Advisor Best Practice Checks (see the Service Limits section)
- Amazon EC2 Service Limits
- What Is Service Quotas?

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas

Plan your network topology

Workloads often exist in multiple environments. These include multiple cloud environments (both publicly accessible and private) and possibly your existing data center infrastructure. Plans must include network considerations, such as intrasystem and intersystem connectivity, public IP address management, private IP address management, and domain name resolution.

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

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

Best practices

- REL02-BP01 Use highly available network connectivity for your workload public endpoints (p. 15)
- REL02-BP02 Provision redundant connectivity between private networks in the cloud and on-premises environments (p. 17)
- REL02-BP03 Ensure IP subnet allocation accounts for expansion and availability (p. 19)
- REL02-BP04 Prefer hub-and-spoke topologies over many-to-many mesh (p. 21)
- REL02-BP05 Enforce non-overlapping private IP address ranges in all private address spaces where they are connected (p. 22)

REL02-BP01 Use highly available network connectivity for your workload public endpoints

These endpoints and the routing to them must be highly available. To achieve this, use highly available DNS, content delivery networks (CDNs), API Gateway, load balancing, or reverse proxies.

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

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

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

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

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

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

Common anti-patterns:

- Using public internet addresses on instances or containers and managing the connectivity to them via DNS.
- Using Internet Protocol addresses instead of domain names for locating services.
- Providing content (web pages, static assets, media files) to a large geographic area and not using a content delivery network.

Benefits of establishing this best practice: By implementing highly available services in your workload, you know that your workload will be available to your users.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Ensure that you have highly available connectivity for users of the workload Amazon Route 53, AWS Global Accelerator, Amazon CloudFront, Amazon API Gateway, and Elastic Load Balancing (ELB) all provide highly available public facing endpoints. You may also choose to evaluate AWS Marketplace software appliances for load-balancing and proxying.

- Ensure that you have a highly available connection to your users.
- Ensure that you are using a highly available DNS to manage the domain names of your application endpoints.
  - If your users access your application via the internet, use service API operations to confirm the correct usage of Internet Gateways. Also confirm that the route tables entries for the subnets hosting your application endpoints are correct.
    - DescribeInternetGateways
    - DescribeRouteTables
- Ensure that you are using a highly available reverse proxy or load balancer in front of your application.
  - If your users access your application via your on-premises environment, ensure that your connectivity between AWS and your on-premises environment is highly available.
  - Use Route 53 to manage your domain names.
    - What is Amazon Route 53?
  - Use a third-party DNS provider that meets your requirements.
  - Use Elastic Load Balancing.
    - What is Elastic Load Balancing?
  - Use an AWS Marketplace appliance that meets your requirements.

Resources

Related documents:

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

- What Is a Transit Gateway?
- What is Amazon CloudFront?
- What is Amazon Route 53?
- What is Elastic Load Balancing?
- Working with Direct Connect Gateways

Related videos:

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

REL02-BP02 Provision redundant connectivity between private networks in the cloud and on-premises environments

Use multiple AWS Direct Connect connections or VPN tunnels between separately deployed private networks. Use multiple Direct Connect locations for high availability. If using multiple AWS Regions, ensure redundancy in at least two of them. You might want to evaluate AWS Marketplace appliances that terminate VPNs. If you use AWS Marketplace appliances, deploy redundant instances for high availability in different Availability Zones.

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

This redundancy addresses possible failures that impact connectivity resiliency:

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

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

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

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

You can also use VPC endpoints to privately connect your VPC to supported AWS services and VPC endpoint services powered by AWS PrivateLink without traversing the public internet. Endpoints are

virtual devices. They are horizontally scaled, redundant, and highly available VPC components. They allow communication between instances in your VPC and services without imposing availability risks or bandwidth constraints on your network traffic.

Common anti-patterns:

- Having only one connectivity provider between your on-site network and AWS.
- Consuming the connectivity capabilities of your AWS Direct Connect connection, but only having one connection.
- Having only one path for your VPN connectivity.

Benefits of establishing this best practice: By implementing redundant connectivity between your cloud environment and you corporate or on-premises environment, you can ensure that the dependent services between the two environments can communicate reliably.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Ensure that you have highly available connectivity between AWS and on-premises environment. Use multiple AWS Direct Connect connections or VPN tunnels between separately deployed private networks. Use multiple Direct Connect locations for high availability. If using multiple AWS Regions, ensure redundancy in at least two of them. You might want to evaluate AWS Marketplace appliances that terminate VPNs. If you use AWS Marketplace appliances, deploy redundant instances for high availability in different Availability Zones.
  - Ensure that you have a redundant connection to your on-premises environment You may need redundant connections to multiple AWS Regions to achieve your availability needs.
    - AWS Direct Connect Resiliency Recommendations
    - Using Redundant Site-to-Site VPN Connections to Provide Failover
      - Use service API operations to identify correct use of Direct Connect circuits.
        - DescribeConnections
        - DescribeConnectionsOnInterconnect
        - DescribeDirectConnectGatewayAssociations
        - DescribeDirectConnectGatewayAttachments
        - DescribeDirectConnectGateways
        - DescribeHostedConnections
        - DescribeInterconnects
      - If only one Direct Connect connection exists or you have none, set up redundant VPN tunnels to your virtual private gateways.
        - What is AWS Site-to-Site VPN?
  - Capture your current connectivity (for example, Direct Connect, virtual private gateways, AWS Marketplace appliances).
    - Use service API operations to query configuration of Direct Connect connections.
      - DescribeConnections
      - DescribeConnectionsOnInterconnect
      - DescribeDirectConnectGatewayAssociations
      - DescribeDirectConnectGatewayAttachments
      - DescribeDirectConnectGateways
      - DescribeHostedConnections
      - DescribeInterconnects
    - Use service API operations to collect virtual private gateways where route tables use them.

    - DescribeVpnGateways
    - DescribeRouteTables
- Use service API operations to collect AWS Marketplace applications where route tables use them.
    - DescribeRouteTables

Resources

Related documents:

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

Related videos:

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

REL02-BP03 Ensure IP subnet allocation accounts for expansion and availability

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

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

- Allow IP address space for more than one VPC per Region.
- Within a VPC, allow space for multiple subnets that span multiple Availability Zones.
- Always leave unused CIDR block space within a VPC for future expansion.
- Ensure that there is IP address space to meet the needs of any transient fleets of EC2 instances that you might use, such as Spot Fleets for machine learning, Amazon EMR clusters, or Amazon Redshift clusters.
- Note that the first four IP addresses and the last IP address in each subnet CIDR block are reserved and not available for your use.
- You should plan on deploying large VPC CIDR blocks. Note that the initial VPC CIDR block allocated to your VPC cannot be changed or deleted, but you can add additional non-overlapping CIDR blocks to the VPC. Subnet IPv4 CIDRs cannot be changed, however IPv6 CIDRs can. Keep in mind that deploying

the largest VPC possible (/16) results in over 65,000 IP addresses. In the base 10.x.x.x IP address space alone, you could provision 255 such VPCs. You should therefore err on the side of being too large rather than too small to make it easier to manage your VPCs.

Common anti-patterns:

- Creating small VPCs.
- Creating small subnets and then having to add subnets to configurations as you grow.
- Incorrectly estimating how many IP addresses a elastic load balancer can use.
- Deploying many high traffic load balancers into the same subnets.

Benefits of establishing this best practice: This ensures that you can accommodate the growth of your workloads and continue to provide availability as you scale up.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Plan your network to accommodate for growth, regulatory compliance, and integration with others. Growth can be underestimated, regulatory compliance can change, and acquisitions or private network connections can be difficult to implement without proper planning.
  - Select relevant AWS accounts and Regions based on your service requirements, latency, regulatory, and disaster recovery (DR) requirements.
  - Identify your needs for regional VPC deployments.
  - Identify the size of the VPCs.
    - Determine if you are going to deploy multi-VPC connectivity.
      - What Is a Transit Gateway?
      - Single Region Multi-VPC Connectivity
    - Determine if you need segregated networking for regulatory requirements.
    - Make VPCs as large as possible. The initial VPC CIDR block allocated to your VPC cannot be changed or deleted, but you can add additional non-overlapping CIDR blocks to the VPC. This however may fragment your address ranges.
    - Make VPCs as large as possible. The initial VPC CIDR block allocated to your VPC cannot be changed or deleted, but you can add additional non-overlapping CIDR blocks to the VPC. This however may fragment your address ranges.

Resources

Related documents:

- APN Partner: partners that can help plan your networking
- AWS Marketplace for Network Infrastructure
- Amazon Virtual Private Cloud Connectivity Options Whitepaper
- Multiple data center HA network connectivity
- Single Region Multi-VPC Connectivity
- What Is Amazon VPC?

Related videos:

- AWS re:Invent 2018: Advanced VPC Design and New Capabilities for Amazon VPC (NET303)

- AWS re:Invent 2019: AWS Transit Gateway reference architectures for many VPCs (NET406-R1)

REL02-BP04 Prefer hub-and-spoke topologies over many-to-many mesh

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

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

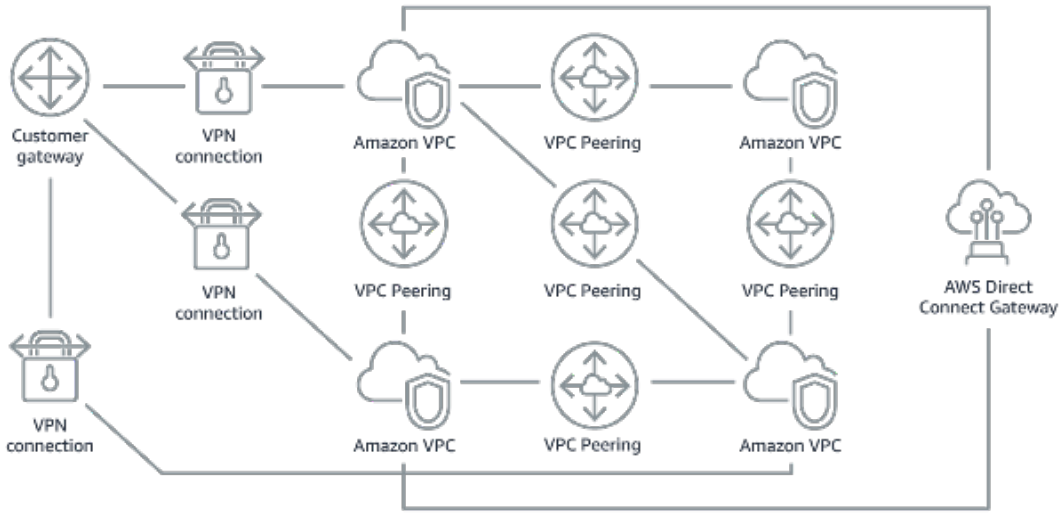


Figure 1: Without AWS Transit Gateway: You need to peer each Amazon VPC to each other and to each onsite location using a VPN connection, which can become complex as it scales.

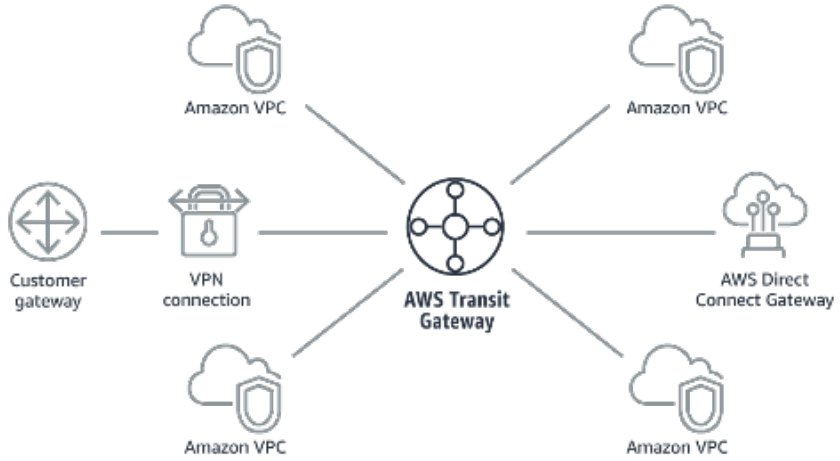


Figure 2: With AWS Transit Gateway: You simply connect each Amazon VPC or VPN to the AWS Transit Gateway and it routes traffic to and from each VPC or VPN.

Common anti-patterns:

- Using VPC peering to connect more than two VPCs.
- Establishing multiple BGP sessions for each VPC to establish connectivity that spans Virtual Private Clouds (VPCs) spread across multiple AWS Regions.

Benefits of establishing this best practice: As the number of networks grows, the complexity of such meshed connections becomes untenable. AWS Transit Gateway provides an easy to maintain hub-and-spoke model, allowing routing of traffic among your multiple networks.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Prefer hub-and-spoke topologies over many-to-many mesh. If more than two network address spaces (VPCs, on-premises networks) are connected via VPC peering, AWS Direct Connect, or VPN, then use a hub-and-spoke model like that provided by AWS Transit Gateway.
  - For only two such networks, you can simply connect them to each other, but as the number of networks grows, the complexity of such meshed connections becomes untenable. AWS Transit Gateway provides an easy to maintain hub-and-spoke model, allowing routing of traffic across your multiple networks.
    - What Is a Transit Gateway?

Resources

Related documents:

- APN Partner: partners that can help plan your networking
- AWS Marketplace for Network Infrastructure
- Multiple data center HA network connectivity
- VPC Endpoints and VPC Endpoint Services (AWS PrivateLink)
- What Is Amazon VPC?
- What Is a Transit Gateway?

Related videos:

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

REL02-BP05 Enforce non-overlapping private IP address ranges in all private address spaces where they are connected

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

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

Common anti-patterns:

- Using the same IP range in your VPC as you have on premises or in your corporate network.
- Not tracking IP ranges of VPCs used to deploy your workloads.

Benefits of establishing this best practice: Active planning of your network will ensure that you do not have multiple occurrences of the same IP address in interconnected networks. This prevents routing problems from occurring in parts of the workload that are using the different applications.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Monitor and manage your CIDR use. Evaluate your potential usage on AWS, add CIDR ranges to existing VPCs, and create VPCs to allow planned growth in usage.
  - Capture current CIDR consumption (for example, VPCs, subnets)
    - Use service API operations to collect current CIDR consumption.
  - Capture your current subnet usage.
    - Use service API operations to collect subnets per VPC in each Region.
      - DescribeSubnets
    - Record the current usage.
    - Determine if you created any overlapping IP ranges.
    - Calculate the spare capacity.
    - Identify overlapping IP ranges. You can either migrate to a new range of addresses or use Network and Port Translation (NAT) appliances from AWS Marketplace if you need to connect the overlapping ranges.

Resources

Related documents:

- APN Partner: partners that can help plan your networking
- AWS Marketplace for Network Infrastructure
- Amazon Virtual Private Cloud Connectivity Options Whitepaper
- Multiple data center HA network connectivity
- What Is Amazon VPC?
- What is IPAM?

Related videos:

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

Workload architecture

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

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

Topics

Design your workload service architecture

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

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

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

Best practices

REL03-BP01 Choose how to segment your workload

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

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

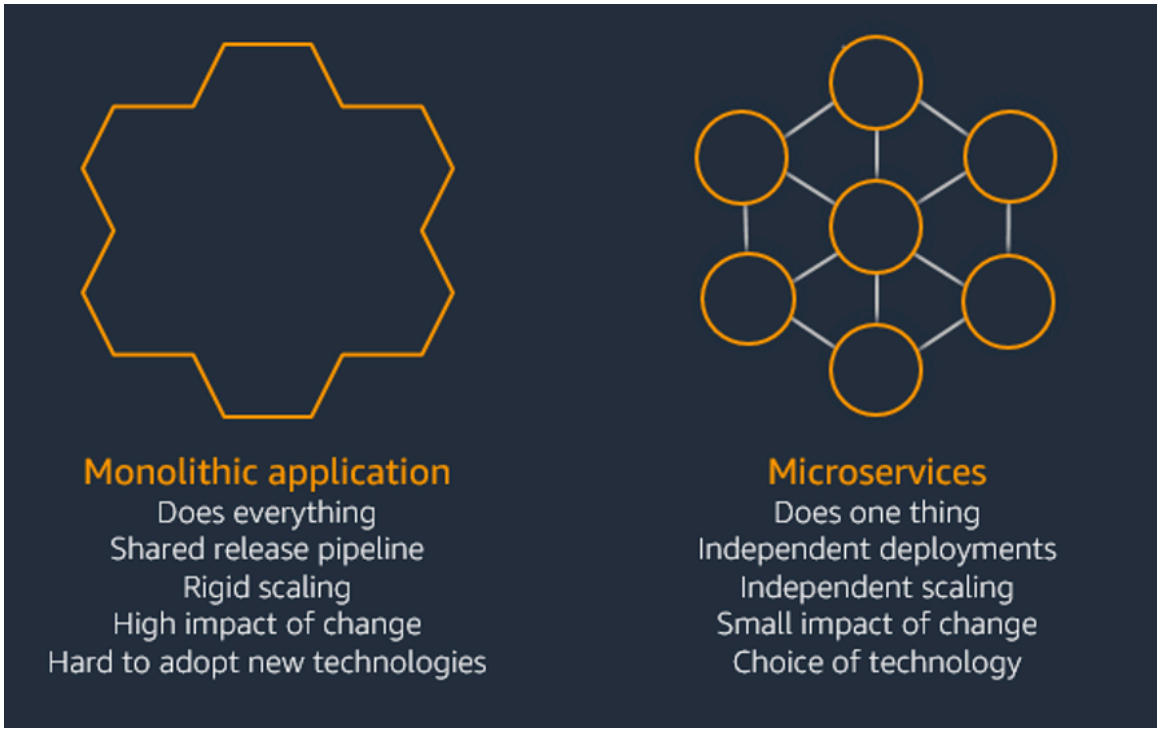


Figure 3: Monolithic architecture versus microservices architecture

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Choose your architecture type based on how you will segment your workload. Choose a service-oriented architecture (SOA) or microservices architecture (or in some cases a monolithic architecture).
  - SOA and microservices offer respectively smaller segmentation, which is preferred as a modern scalable and reliable architecture. However, if you have a good reason to choose a monolithic architecture, then you must ensure it's modular and has the ability to ultimately evolve to SOA or microservices as your product scales with user adoption.
  - SOA can be a good compromise for achieving smaller segmentation while avoiding some of the complexities of microservices.
    - Microservice Trade-Offs
  - If your workload is amenable to it, and your organization can support it, you should use a microservices architecture to achieve the best agility and reliability.
    - Implementing Microservices on AWS
  - AWS App Mesh can be used with service-oriented architectures to provide reliable discovery and access of services.
    - What is AWS App Mesh?

Resources

Related documents:

- Amazon API Gateway: Configuring a REST API Using OpenAPI
- Bounded Context (a central pattern in Domain-Driven Design)
- Implementing Microservices on AWS
- Microservice Trade-Offs
- Microservices - a definition of this new architectural term
- Microservices on AWS
- What is AWS App Mesh?

REL03-BP02 Build services focused on specific business domains and functionality

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Design your workload based on your business domains and their respective functionality. Focusing on specific functionality enables you to differentiate the reliability requirements of different services, and target investments more specifically. A concise business problem and having a small team associated with each service also enables easier organizational scaling.
  - Perform Domain Analysis to map out a domain-driven design (DDD) for your workload. Then you can choose an architecture type to meet your workload's needs.
    - How to break a Monolith into Microservices
    - Getting Started with DDD when Surrounded by Legacy Systems
    - Eric Evans "Domain-Driven Design: Tackling Complexity in the Heart of Software"
    - Implementing Microservices on AWS

- Decompose your services into smallest possible components. With microservices architecture you can separate your workload into components with the minimal functionality to enable organizational scaling and agility.
  - Define the API for the workload and its design goals, limits, and any other considerations for use.
    - Define the API.
      - The API definition should allow for growth and additional parameters.
    - Define the designed availabilities.
      - Your API may have multiple design goals for different features.
    - Establish limits
      - Use testing to define the limits of your workload capabilities.

Resources

Related documents:

- Amazon API Gateway: Configuring a REST API Using OpenAPI
- Bounded Context (a central pattern in Domain-Driven Design)
- Eric Evans “Domain-Driven Design: Tackling Complexity in the Heart of Software”
- Getting Started with DDD when Surrounded by Legacy Systems
- How to break a Monolith into Microservices
- Implementing Microservices on AWS
- Microservice Trade-Offs
- Microservices - a definition of this new architectural term
- Microservices on AWS

REL03-BP03 Provide service contracts per API

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

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

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

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. It handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, authorization and access control, monitoring, and API version management. Using OpenAPI Specification

(OAS), formerly known as the Swagger Specification, you can define your API contract and import it into API Gateway. With API Gateway, you can then version and deploy the APIs.

Level of risk exposed if this best practice is not established: Low

Implementation guidance

- Provide service contracts per API Service contracts are documented agreements between teams on service integration and include a machine-readable API definition, rate limits, and performance expectations.
  - Amazon API Gateway: Configuring a REST API Using OpenAPI
    - A versioning strategy allows clients to continue using the existing API and migrate their applications to the newer API when they are ready.
    - Amazon API Gateway is a fully managed service that makes it easy for developers to create APIs at any scale. Using the OpenAPI Specification (OAS), formerly known as the Swagger Specification, you can define your API contract and import it into API Gateway. With API Gateway, you can then version and deploy the APIs.

Resources

Related documents:

- Amazon API Gateway: Configuring a REST API Using OpenAPI
- Bounded Context (a central pattern in Domain-Driven Design)
- Implementing Microservices on AWS
- Microservice Trade-Offs
- Microservices - a definition of this new architectural term
- Microservices on AWS

Design interactions in a distributed system to prevent failures

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

Best practices

- REL04-BP01 Identify which kind of distributed system is required (p. 29)
- REL04-BP02 Implement loosely coupled dependencies (p. 30)
- REL04-BP03 Do constant work (p. 32)
- REL04-BP04 Make all responses idempotent (p. 33)

REL04-BP01 Identify which kind of distributed system is required

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Identify which kind of distributed system is required. Challenges with distributed systems involved latency, scaling, understanding networking APIs, marshalling and unmarshalling data, and the complexity of algorithms such as Paxos. As the systems grow larger and more distributed, what had been theoretical edge cases turn into regular occurrences.
  - The Amazon Builders' Library: Challenges with distributed systems
    - Hard real-time distributed systems require responses to be given synchronously and rapidly.
    - Soft real-time systems have a more generous time window of minutes or greater for response.
    - Offline systems handle responses through batch or asynchronous processing.
    - Hard real-time distributed systems have the most stringent reliability requirements.

Resources

Related documents:

- Amazon EC2: Ensuring Idempotency
- The Amazon Builders' Library: Challenges with distributed systems
- The Amazon Builders' Library: Reliability, constant work, and a good cup of coffee
- What Is Amazon EventBridge?
- What Is Amazon Simple Queue Service?

Related videos:

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

REL04-BP02 Implement loosely coupled dependencies

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

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

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

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

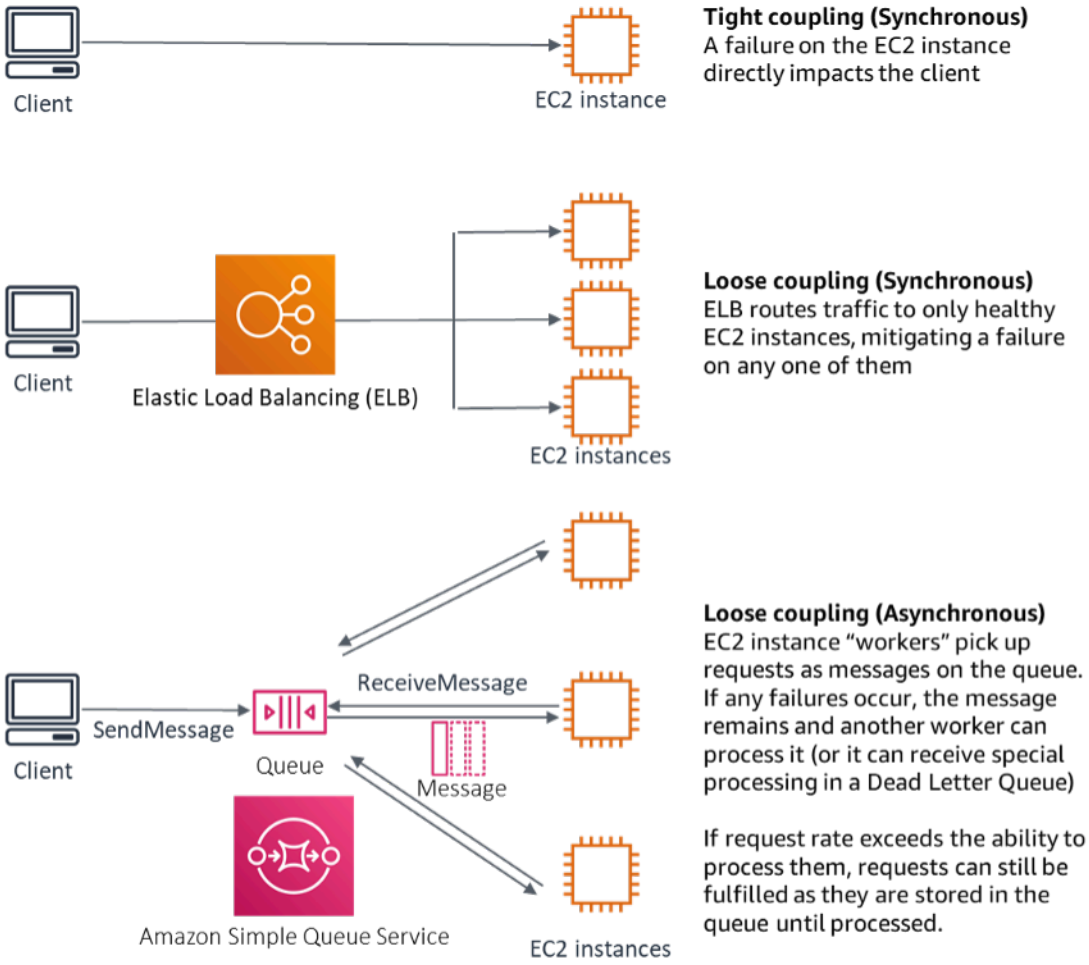


Figure 4: Dependencies such as queuing systems and load balancers are loosely coupled

Amazon SQS queues and Elastic Load Balancers are just two ways to add an intermediate layer for loose coupling. Event-driven architectures can also be built in the AWS Cloud using Amazon EventBridge, which can abstract clients (event producers) from the services they rely on (event consumers). Amazon Simple Notification Service (Amazon SNS) is an effective solution when you need high-throughput, push-based, many-to-many messaging. Using Amazon SNS topics, your publisher systems can fan out messages to a large number of subscriber endpoints for parallel processing.

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

Common anti-patterns:

- Deploying a singleton as part of a workload.
- Directly invoking APIs between workload tiers with no capability of failover or asynchronous processing of the request.

Benefits of establishing this best practice: Loose coupling helps isolate behavior of a component from other components that depend on it, increasing resiliency and agility. Failure in one component is isolated from others.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Implement loosely coupled dependencies. Dependencies such as queuing systems, streaming systems, workflows, and load balancers are loosely coupled. Loose coupling helps isolate behavior of a component from other components that depend on it, increasing resiliency and agility.
  - AWS re:Invent 2019: Moving to event-driven architectures (SVS308)
  - What Is Amazon EventBridge?
  - What Is Amazon Simple Queue Service?
    - Amazon EventBridge allows you to build event driven architectures, which are loosely coupled and distributed.
      - AWS New York Summit 2019: Intro to Event-driven Architectures and Amazon EventBridge (MAD205)
    - If changes to one component force other components that rely on it to also change, then they are tightly coupled. Loose coupling breaks this dependency so that dependency components only need to know the versioned and published interface.
    - Make component interactions asynchronous where possible. This model is suitable for any interaction that does not need an immediate response and where an acknowledgement that a request has been registered will suffice.
      - AWS re:Invent 2019: Scalable serverless event-driven applications using Amazon SQS and Lambda (API304)

Resources

Related documents:

- AWS re:Invent 2019: Moving to event-driven architectures (SVS308)
- Amazon EC2: Ensuring Idempotency

- The Amazon Builders' Library: Challenges with distributed systems
- The Amazon Builders' Library: Reliability, constant work, and a good cup of coffee
- What Is Amazon EventBridge?
- What Is Amazon Simple Queue Service?

Related videos:

- AWS New York Summit 2019: Intro to Event-driven Architectures and Amazon EventBridge (MAD205)
- AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small ARC337 (includes loose coupling, constant work, static stability)
- AWS re:Invent 2019: Moving to event-driven architectures (SVS308)
- AWS re:Invent 2019: Scalable serverless event-driven applications using Amazon SQS and Lambda (API304)

REL04-BP03 Do constant work

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

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

Level of risk exposed if this best practice is not established: Low

Implementation guidance

- Do constant work so that systems do not fail when there are large, rapid changes in load.
- Implement loosely coupled dependencies. Dependencies such as queuing systems, streaming systems, workflows, and load balancers are loosely coupled. Loose coupling helps isolate behavior of a component from other components that depend on it, increasing resiliency and agility.
  - The Amazon Builders' Library: Reliability, constant work, and a good cup of coffee
  - AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small ARC337 (includes constant work)
    - For the example of a health check system monitoring 100,000 servers, engineer workloads so that payload sizes remain constant regardless of number of successes or failures.

Resources

Related documents:

- Amazon EC2: Ensuring Idempotency
- The Amazon Builders' Library: Challenges with distributed systems

- The Amazon Builders' Library: Reliability, constant work, and a good cup of coffee

Related videos:

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

REL04-BP04 Make all responses idempotent

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

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Make all responses idempotent. An idempotent service promises that each request is completed exactly once, such that making multiple identical requests has the same effect as making a single request.
  - Clients can issue API requests with an idempotency token—the same token is used whenever the request is repeated. An idempotent service API uses the token to return a response identical to the response that was returned the first time that the request was completed.
    - Amazon EC2: Ensuring Idempotency

Resources

Related documents:

- Amazon EC2: Ensuring Idempotency
- The Amazon Builders' Library: Challenges with distributed systems
- The Amazon Builders' Library: Reliability, constant work, and a good cup of coffee

Related videos:

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

- AWS re:Invent 2019: Moving to event-driven architectures (SVS308)

Design interactions in a distributed system to mitigate or withstand failures

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

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

Best practices

- REL05-BP01 Implement graceful degradation to transform applicable hard dependencies into soft dependencies (p. 34)
- REL05-BP02 Throttle requests (p. 36)
- REL05-BP03 Control and limit retry calls (p. 37)
- REL05-BP04 Fail fast and limit queues (p. 38)
- REL05-BP05 Set client timeouts (p. 39)
- REL05-BP06 Make services stateless where possible (p. 40)
- REL05-BP07 Implement emergency levers (p. 41)

REL05-BP01 Implement graceful degradation to transform applicable hard dependencies into soft dependencies

When a component's dependencies are unhealthy, the component itself can still function, although in a degraded manner. For example, when a dependency call fails, failover to a predetermined static response.

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

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

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

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

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

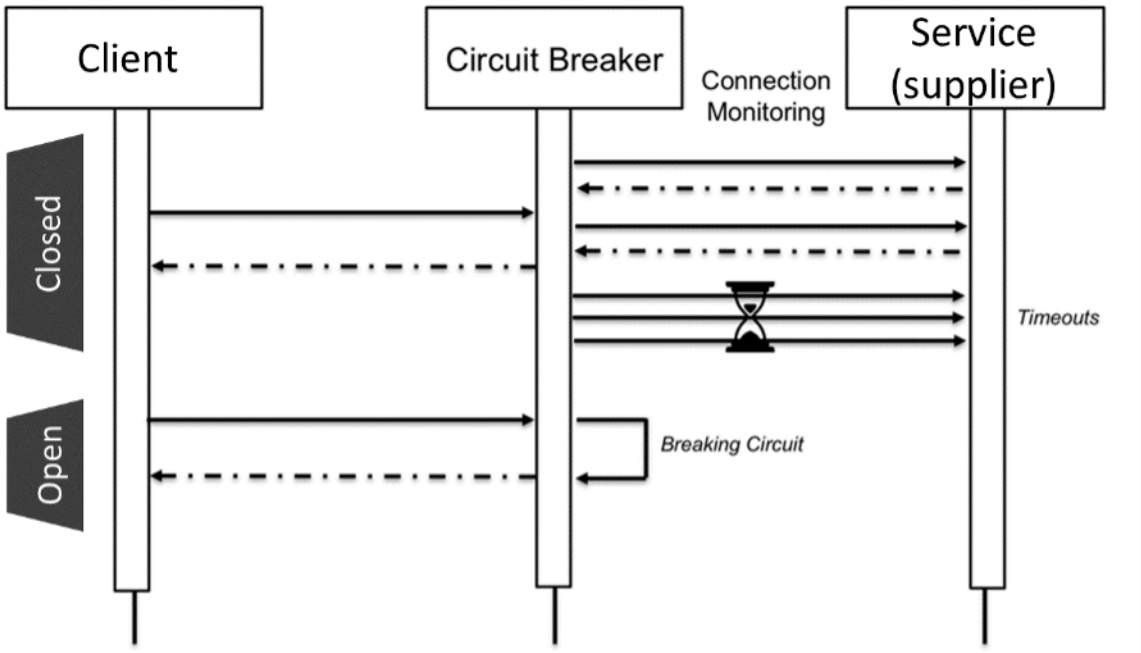


Figure 6: Circuit breaker showing closed and open states.

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

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

- By returning a static response, your workload mitigates failures that occur in its dependencies.
  - Well-Architected lab: Level 300: Implementing Health Checks and Managing Dependencies to Improve Reliability
- Detect when the retry operation is likely to fail, and prevent your client from making failed calls with the circuit breaker pattern.
  - CircuitBreaker

Resources

Related documents:

- Amazon API Gateway: Throttle API Requests for Better Throughput
- CircuitBreaker (summarizes Circuit Breaker from “Release It!” book)
- Error Retries and Exponential Backoff in AWS
- Michael Nygard “Release It! Design and Deploy Production-Ready Software”
- The Amazon Builders' Library: Avoiding fallback in distributed systems
- The Amazon Builders' Library: Avoiding insurmountable queue backlogs
- The Amazon Builders' Library: Caching challenges and strategies
- The Amazon Builders' Library: Timeouts, retries, and backoff with jitter

Related videos:

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

Related examples:

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

REL05-BP02 Throttle requests

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

Your services should be designed to handle a known capacity of requests that each node or cell can process. This capacity 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 or cell that might have available capacity. Amazon API Gateway provides methods for throttling requests. Amazon SQS and Amazon Kinesis can buffer requests, smooth out the request rate, and alleviate the need for throttling for requests that can be addressed asynchronously.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Throttle requests. This is a mitigation pattern to respond to an unexpected increase in demand. Some requests are honored but those over a defined limit are rejected and return a message indicating they

have been throttled. The expectation on clients is that they will back off and abandon the request or try again at a slower rate.

- Use Amazon API Gateway
  - Throttle API Requests for Better Throughput

Resources

Related documents:

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

Related videos:

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

REL05-BP03 Control and limit retry calls

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

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

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

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

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Control and limit retry calls. Use exponential backoff to retry after progressively longer intervals. Introduce jitter to randomize those retry intervals, and limit the maximum number of retries.
  - Error Retries and Exponential Backoff in AWS
    - Amazon SDKs implement retries and exponential backoff by default. Implement similar logic in your dependency layer when calling your own dependent services. Decide what the timeouts are and when to stop retrying based on your use case.

Resources

Related documents:

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

Related videos:

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

REL05-BP04 Fail fast and limit queues

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

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Fail fast and limit queues. If the workload is unable to respond successfully to a request, then fail fast. This allows the releasing of resources associated with a request, and permits the service to recover if it's running out of resources. If the workload is able to respond successfully but the rate of requests is too high, then use a queue to buffer requests instead. However, do not allow long queues that can result in serving stale requests that the client has already given up on.
  - Implement fail fast when service is under stress.
    - Fail Fast
  - Limit queues In a queue-based system, when processing stops but messages keep arriving, the message debt can accumulate into a large backlog, driving up processing time. Work can be

completed too late for the results to be useful, essentially causing the availability hit that queueing was meant to guard against.

- The Amazon Builders' Library: Avoiding insurmountable queue backlogs

Resources

Related documents:

- Error Retries and Exponential Backoff in AWS
- Fail Fast
- The Amazon Builders' Library: Avoiding fallback in distributed systems
- The Amazon Builders' Library: Avoiding insurmountable queue backlogs
- The Amazon Builders' Library: Caching challenges and strategies
- The Amazon Builders' Library: Timeouts, retries, and backoff with jitter

Related videos:

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

REL05-BP05 Set client timeouts

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

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

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

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

Resources

Related documents:

- AWS SDK: Retries and Timeouts
- Amazon API Gateway: Throttle API Requests for Better Throughput
- Error Retries and Exponential Backoff in AWS
- The Amazon Builders' Library: Timeouts, retries, and backoff with jitter

Related videos:

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

REL05-BP06 Make services stateless where possible

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

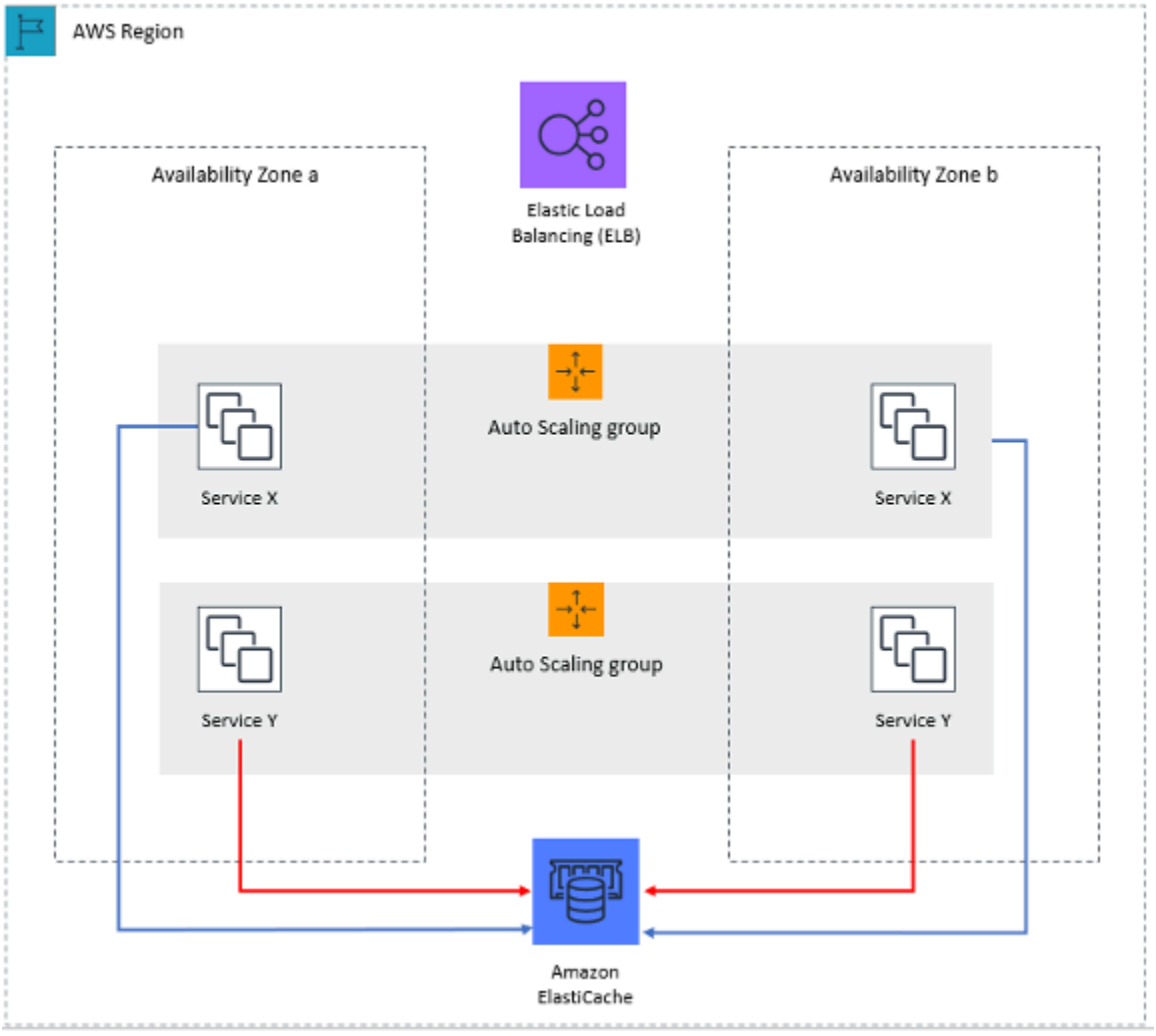


Figure 7: In this stateless web application, session state is offloaded to Amazon ElastiCache.

When users or services interact with an application, they often perform a series of interactions that form a session. A session is unique data for users that persists between requests while they use

the application. A stateless application is an application that does not need knowledge of previous interactions and does not store session information.

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

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Make your applications stateless. Stateless applications enable horizontal scaling and are tolerant to the failure of an individual node.
  - Remove state that could actually be stored in request parameters.
  - After examining whether the state is required, move any state tracking to a resilient multi-zone cache or data store like Amazon ElastiCache, Amazon RDS, Amazon DynamoDB, or a third-party distributed data solution. Store a state that could not be moved to resilient data stores.
    - Some data (like cookies) can be passed in headers or query parameters.
    - Refactor to remove state that can be quickly passed in requests.
    - Some data may not actually be needed per request and can be retrieved on demand.
    - Remove data that can be asynchronously retrieved.
    - Decide on a data store that meets the requirements for a required state.
    - Consider a NoSQL database for non-relational data.

Resources

Related documents:

- The Amazon Builders' Library: Avoiding fallback in distributed systems
- The Amazon Builders' Library: Avoiding insurmountable queue backlogs
- The Amazon Builders' Library: Caching challenges and strategies

REL05-BP07 Implement emergency levers

Emergency levers are rapid processes that can mitigate availability impact on your workload.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Implement emergency levers. These are rapid processes that may mitigate availability impact on your workload. They can be operated in the absence of a root cause. An ideal emergency lever reduces the cognitive burden on the resolvers to zero by providing fully deterministic activation and deactivation criteria. Levers are often manual, but they can also be automated
  - Example levers include
    - Block all robot traffic
    - Serve static pages instead of dynamic ones
    - Reduce frequency of calls to a dependency

    - Throttle calls from dependencies
- Tips for implementing and using emergency levers
    - When levers are activated, do LESS, not more
    - Keep it simple, avoid bimodal behavior
    - Test your levers periodically
- These are examples of actions that are NOT emergency levers
    - Add capacity
    - Call up service owners of clients that depend on your service and ask them to reduce calls
    - Making a change to code and releasing it

Change management

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

The following sections explain the best practices for change management.

Topics

Monitor workload resources

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

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

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

Monitoring at AWS consists of four distinct phases:

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

Best practices

- REL06-BP04 Automate responses (Real-time processing and alarming) (p. 48)
- REL06-BP05 Analytics (p. 49)
- REL06-BP06 Conduct reviews regularly (p. 50)
- REL06-BP07 Monitor end-to-end tracing of requests through your system (p. 51)

REL06-BP01 Monitor all components for the workload (Generation)

Monitor the components of the workload with Amazon CloudWatch or third-party tools. Monitor AWS services with AWS Health Dashboard.

All components of your workload should be monitored, including the front-end, business logic, and storage tiers. Define key metrics, describe how to extract them from logs (if necessary), and set thresholds for triggering corresponding alarm events. Ensure metrics are relevant to the key performance indicators (KPIs) of your workload, and use metrics and logs to identify early warning signs of service degradation. For example, a metric related to business outcomes such as the number of orders successfully processed per minute, can indicate workload issues faster than technical metric, such as CPU Utilization. Use AWS Health Dashboard for a personalized view into the performance and availability of the AWS services underlying your AWS resources.

Monitoring in the cloud offers new opportunities. Most cloud providers have developed customizable hooks and can deliver insights to help you monitor multiple layers of your workload. AWS services such as Amazon CloudWatch apply statistical and machine learning algorithms to continually analyze metrics of systems and applications, determine normal baselines, and surface anomalies with minimal user intervention. Anomaly detection algorithms account for the seasonality and trend changes of metrics.

AWS makes an abundance of monitoring and log information available for consumption that can be used to define workload-specific metrics, change-in-demand processes, and adopt machine learning techniques regardless of ML expertise.

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

Desired Outcome:

Collect and use critical metrics from all components of the workload to ensure workload reliability and optimal user experience. Detecting that a workload is not achieving business outcomes allows you to quickly declare a disaster and recover from an incident.

Common anti-patterns:

- Only monitoring external interfaces to your workload.
- Not generating any workload-specific metrics and only relying on metrics provided to you by the AWS services your workload uses.
- Only using technical metrics in your workload and not monitoring any metrics related to non-technical KPIs the workload contributes to.
- Relying on production traffic and simple health checks to monitor and evaluate workload state.

Benefits of establishing this best practice: Monitoring at all tiers in your workload enables you to more rapidly anticipate and resolve problems in the components that comprise the workload.

Level of risk exposed if this best practice is not established: High

Implementation guidance

1. **Enable logging where available.** Monitoring data should be obtained from all components of the workloads. Turn on additional logging, such as S3 Access Logs, and enable your workload to log workload specific data. Collect metrics for CPU, network I/O, and disk I/O averages from services such as Amazon ECS, Amazon EKS, Amazon EC2, Elastic Load Balancing, AWS Auto Scaling, and Amazon EMR. See AWS Services That Publish CloudWatch Metrics for a list of AWS services that publish metrics to CloudWatch.
2. **Review all default metrics and explore any data collection gaps.** Every service generates default metrics. Collecting default metrics allows you to better understand the dependencies between workload components, and how component reliability and performance affect the workload. You can also create and publish your own metrics to CloudWatch using the AWS CLI or an API. This
3. **Evaluate all the metrics to decide which ones to alert on for each AWS service in your workload.** You may choose to select a subset of metrics that have a major impact on workload reliability. Focusing on critical metrics and threshold allows you to refine the number of alerts and can help minimize false-positives.
4. **Define alerts and the recovery process for your workload after the alert is triggered.** Defining alerts allows you to quickly notify, escalate, and follow steps necessary to recover from an incident and meet your prescribed Recovery Time Objective (RTO). You can use Amazon CloudWatch Alarms to invoke automated workflows and initiate recovery procedures based on defined thresholds.
5. **Explore use of synthetic transactions to collect relevant data about workloads state.** Synthetic monitoring follows the same routes and perform the same actions as a customer, which makes it possible for you to continually verify your customer experience even when you don't have any customer traffic on your workloads. By using synthetic transactions, you can discover issues before your customers do.

Resources

Related best practices:

- REL11-BP03 Automate healing on all layers (p. 87)

Related documents:

- Getting started with your AWS Health Dashboard – Your account health
- AWS Services That Publish CloudWatch Metrics
- Access Logs for Your Network Load Balancer
- Access logs for your application load balancer
- Accessing Amazon CloudWatch Logs for AWS Lambda
- Amazon S3 Server Access Logging
- Enable Access Logs for Your Classic Load Balancer
- Exporting log data to Amazon S3
- Install the CloudWatch agent on an Amazon EC2 instance
- Publishing Custom Metrics
- Using Amazon CloudWatch Dashboards

- Using Amazon CloudWatch Metrics
- Using Canaries (Amazon CloudWatch Synthetics)
- What are Amazon CloudWatch Logs?

User guides:

- Creating a trail
- Monitoring memory and disk metrics for Amazon EC2 Linux instances
- Using CloudWatch Logs with container instances
- VPC Flow Logs
- What is Amazon DevOps Guru?
- What is AWS X-Ray?

Related blogs:

- Debugging with Amazon CloudWatch Synthetics and AWS X-Ray

Related examples and workshops:

- AWS Well-Architected Labs: Operational Excellence - Dependency Monitoring
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility
- Observability workshop

REL06-BP02 Define and calculate metrics (Aggregation)

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Define and calculate metrics (Aggregation). Store log data and apply filters where necessary to calculate metrics, such as counts of a specific log event, or latency calculated from log event timestamps
  - Metric filters define the terms and patterns to look for in log data as it is sent to CloudWatch Logs. CloudWatch Logs uses these metric filters to turn log data into numerical CloudWatch metrics that you can graph or set an alarm on.
    - Searching and Filtering Log Data
  - Use a trusted third party to aggregate logs.
    - Follow the instructions of the third party. Most third-party products integrate with CloudWatch and Amazon S3.

- Some AWS services can publish logs directly to Amazon S3. If your main requirement for logs is storage in Amazon S3, you can easily have the service producing the logs send them directly to Amazon S3 without setting up additional infrastructure.
  - Sending Logs Directly to Amazon S3

Resources

Related documents:

- Amazon CloudWatch Logs Insights Sample Queries
- Debugging with Amazon CloudWatch Synthetics and AWS X-Ray
- One Observability Workshop
- Searching and Filtering Log Data
- Sending Logs Directly to Amazon S3
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility

REL06-BP03 Send notifications (Real-time processing and alarming)

Organizations that need to know, receive notifications when significant events occur.

Alerts can 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.

Common anti-patterns:

- Configuring alarms at too low of threshold, causing too many notifications to be sent.
- Not archiving alarms for future exploration.

Benefits of establishing this best practice: Notifications on events (even those that can be responded to and automatically resolved) allow you to have a record of events and potentially address them in a different manner in the future.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Perform real-time processing and alarming. Organizations that need to know, receive notifications when significant events occur
  - Amazon CloudWatch dashboards are customizable home pages in the CloudWatch console that you can use to monitor your resources in a single view, even those resources that are spread across different Regions.
    - Using Amazon CloudWatch Dashboards
  - Create an alarm when the metric surpasses a limit.
    - Using Amazon CloudWatch Alarms

Resources

Related documents:

- One Observability Workshop
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility
- Using Amazon CloudWatch Alarms
- Using Amazon CloudWatch Dashboards
- Using Amazon CloudWatch Metrics

REL06-BP04 Automate responses (Real-time processing and alarming)

Use automation to take action when an event is detected, for example, to replace failed components.

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

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Use Amazon DevOps Guru to perform automated actions. Amazon DevOps Guru can automatically monitor application resources for anomalous behavior and deliver targeted recommendations to speed up problem identification and remediation times.
  - What is Amazon DevOps Guru?
- Use AWS Systems Manager to perform automated actions. AWS Config continually monitors and records your AWS resource configurations, and can trigger AWS Systems Manager Automation to remediate issues.
  - AWS Systems Manager Automation
    - Create and use Systems Manager Automation documents. These define the actions that Systems Manager performs on your managed instances and other AWS resources when an automation process runs.
    - Working with Automation Documents (Playbooks)
- Amazon CloudWatch sends alarm state change events to Amazon EventBridge. Create EventBridge rules to automate responses.
  - Creating an EventBridge Rule That Triggers on an Event from an AWS Resource
- Create and execute a plan to automate responses.
  - Inventory all your alert response procedures. You must plan your alert responses before you rank the tasks.
  - Inventory all the tasks with specific actions that must be taken. Most of these actions are documented in runbooks. You must also have playbooks for alerts of unexpected events.
  - Examine the runbooks and playbooks for all automatable actions. In general, if an action can be defined, it most likely can be automated.
  - Rank the error-prone or time-consuming activities first. It is most beneficial to remove sources of errors and reduce time to resolution.
  - Establish a plan to complete automation. Maintain an active plan to automate and update the automation.

- Examine manual requirements for opportunities for automation. Challenge your manual process for opportunities to automate.

Resources

Related documents:

- AWS Systems Manager Automation
- Creating an EventBridge Rule That Triggers on an Event from an AWS Resource
- One Observability Workshop
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility
- What is Amazon DevOps Guru?
- Working with Automation Documents (Playbooks)

REL06-BP05 Analytics

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

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

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

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- CloudWatch Logs Insights enables you to interactively search and analyze your log data in Amazon CloudWatch Logs.
  - Analyzing Log Data with CloudWatch Logs Insights
  - Amazon CloudWatch Logs Insights Sample Queries
- Use Amazon CloudWatch Logs send logs to Amazon S3 where you can use or Amazon Athena to query the data.
  - How do I analyze my Amazon S3 server access logs using Athena?
    - Create an S3 lifecycle policy for your server access logs bucket. Configure the lifecycle policy to periodically remove log files. Doing so reduces the amount of data that Athena analyzes for each query.
      - How Do I Create a Lifecycle Policy for an S3 Bucket?

Resources

Related documents:

- Amazon CloudWatch Logs Insights Sample Queries
- Analyzing Log Data with CloudWatch Logs Insights
- Debugging with Amazon CloudWatch Synthetics and AWS X-Ray
- How Do I Create a Lifecycle Policy for an S3 Bucket?
- How do I analyze my Amazon S3 server access logs using Athena?
- One Observability Workshop
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility

REL06-BP06 Conduct reviews regularly

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

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

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

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

At AWS, we conduct a weekly meeting to review operational performance and to share learnings between teams. Because there are so many teams in AWS, we created The Wheel to randomly pick a workload to review. Establishing a regular cadence for operational performance reviews and knowledge sharing enhances your ability to achieve higher performance from your operational teams.

Common anti-patterns:

- Collecting only default metrics.
- Setting a monitoring strategy and never reviewing it.
- Not discussing monitoring when major changes are deployed.

Benefits of establishing this best practice: Regularly reviewing your monitoring enables the anticipation of potential problems, instead of reacting to notifications when an anticipated problem actually occurs.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Create multiple dashboards for the workload. You must have a top-level dashboard that contains the key business metrics, as well as the technical metrics you have identified to be the most relevant to the projected health of the workload as usage varies. You should also have dashboards for various application tiers and dependencies that can be inspected.

  - Using Amazon CloudWatch Dashboards
- Schedule and conduct regular reviews of the workload dashboards. Conduct regular inspection of the dashboards. You may have different cadences for the depth at which you inspect.
  - Inspect for trends in the metrics. Compare the metric values to historic values to see if there are trends that may indicate that something that needs investigation. Examples of this include: increasing latency, decreasing primary business function, and increasing failure responses.
  - Inspect for outliers/anomalies in your metrics. Averages or medians can mask outliers and anomalies. Look at the highest and lowest values during the time frame and investigate the causes of extreme scores. As you continue to eliminate these causes, lowering your definition of extreme allows you to continue to improve the consistency of your workload performance.
  - Look for sharp changes in behavior. An immediate change in quantity or direction of a metric may indicate that there has been a change in the application, or external factors that you may need to add additional metrics to track.

Resources

Related documents:

- Amazon CloudWatch Logs Insights Sample Queries
- Debugging with Amazon CloudWatch Synthetics and AWS X-Ray
- One Observability Workshop
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility
- Using Amazon CloudWatch Dashboards

REL06-BP07 Monitor end-to-end tracing of requests through your system

Use AWS X-Ray or third-party tools so that developers can more easily analyze and debug distributed systems to understand how their applications and its underlying services are performing.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Monitor end-to-end tracing of requests through your system. AWS X-Ray is a service that collects data about requests that your application serves, and provides tools you can use to view, filter, and gain insights into that data to identify issues and opportunities for optimization. For any traced request to your application, you can see detailed information not only about the request and response, but also about calls that your application makes to downstream AWS resources, microservices, databases, and web APIs.
  - What is AWS X-Ray?
  - Debugging with Amazon CloudWatch Synthetics and AWS X-Ray

Resources

Related documents:

- Debugging with Amazon CloudWatch Synthetics and AWS X-Ray
- One Observability Workshop
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility

- Using Canaries (Amazon CloudWatch Synthetics)
- What is AWS X-Ray?

Design your workload to adapt to changes in demand

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

Best practices

- REL07-BP01 Use automation when obtaining or scaling resources (p. 52)
- REL07-BP02 Obtain resources upon detection of impairment to a workload (p. 54)
- REL07-BP03 Obtain resources upon detection that more resources are needed for a workload (p. 55)
- REL07-BP04 Load test your workload (p. 56)

REL07-BP01 Use automation when obtaining or scaling resources

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

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

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

When scaling EC2 instances, ensure that you use multiple Availability Zones (preferably at least three) and add or remove capacity to maintain balance across these Availability Zones. ECS tasks or Kubernetes pods (when using Amazon Elastic Kubernetes Service) should also be distributed across multiple Availability Zones.

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

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

Configure and use Amazon CloudFront or a trusted content delivery network (CDN). A CDN can provide faster end-user response times and can serve requests for content from cache, therefore reducing the need to scale your workload.

Common anti-patterns:

- Implementing Auto Scaling groups for automated healing, but not implementing elasticity.

- Using automatic scaling to respond to large increases in traffic.
- Deploying highly stateful applications, eliminating the option of elasticity.

Benefits of establishing this best practice: Automation removes the potential for manual error in deploying and decommissioning resources. Automation removes the risk of cost overruns and denial of service due to slow response on needs for deployment or decommissioning.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Configure and use AWS Auto Scaling. This monitors your applications and automatically adjusts capacity to maintain steady, predictable performance at the lowest possible cost. Using AWS Auto Scaling, you can setup application scaling for multiple resources across multiple services.
  - What is AWS Auto Scaling?
    - Configure Auto Scaling on your Amazon EC2 instances and Spot Fleets, Amazon ECS tasks, Amazon DynamoDB tables and indexes, Amazon Aurora Replicas, and AWS Marketplace appliances as applicable.
      - Managing throughput capacity automatically with DynamoDB Auto Scaling
        - Use service API operations to specify the alarms, scaling policies, warm up times, and cool down times.
- Use Elastic Load Balancing. Load balancers can distribute load by path or by network connectivity.
  - What is Elastic Load Balancing?
    - Application Load Balancers can distribute load by path.
      - What is an Application Load Balancer?
        - Configure an Application Load Balancer to distribute traffic to different workloads based on the path under the domain name.
        - Application Load Balancers can be used to distribute loads in a manner that integrates with AWS Auto Scaling to manage demand.
          - Using a load balancer with an Auto Scaling group
    - Network Load Balancers can distribute load by connection.
      - What is a Network Load Balancer?
        - Configure a Network Load Balancer to distribute traffic to different workloads using TCP, or to have a constant set of IP addresses for your workload.
        - Network Load Balancers can be used to distribute loads in a manner that integrates with AWS Auto Scaling to manage demand.
- Use a highly available DNS provider. DNS names allow your users to enter names instead of IP addresses to access your workloads and distributes this information to a defined scope, usually globally for users of the workload.
  - Use Amazon Route 53 or a trusted DNS provider.
    - What is Amazon Route 53?
  - Use Route 53 to manage your CloudFront distributions and load balancers.
    - Determine the domains and subdomains you are going to manage.
    - Create appropriate record sets using ALIAS or CNAME records.
      - Working with records
- Use the AWS global network to optimize the path from your users to your applications. AWS Global Accelerator continually monitors the health of your application endpoints and redirects traffic to healthy endpoints in less than 30 seconds.
  - AWS Global Accelerator is a service that improves the availability and performance of your applications with local or global users. It provides static IP addresses that act as a fixed entry point

to your application endpoints in a single or multiple AWS Regions, such as your Application Load Balancers, Network Load Balancers or Amazon EC2 instances.
    - What Is AWS Global Accelerator?
- Configure and use Amazon CloudFront or a trusted content delivery network (CDN). A content delivery network can provide faster end-user response times and can serve requests for content that may cause unnecessary scaling of your workloads.
    - What is Amazon CloudFront?
        - Configure Amazon CloudFront distributions for your workloads, or use a third-party CDN.
            - You can limit access to your workloads so that they are only accessible from CloudFront by using the IP ranges for CloudFront in your endpoint security groups or access policies.

Resources

Related documents:

- APN Partner: partners that can help you create automated compute solutions
- AWS Auto Scaling: How Scaling Plans Work
- AWS Marketplace: products that can be used with auto scaling
- Managing Throughput Capacity Automatically with DynamoDB Auto Scaling
- Using a load balancer with an Auto Scaling group
- What Is AWS Global Accelerator?
- What Is Amazon EC2 Auto Scaling?
- What is AWS Auto Scaling?
- What is Amazon CloudFront?
- What is Amazon Route 53?
- What is Elastic Load Balancing?
- What is a Network Load Balancer?
- What is an Application Load Balancer?
- Working with records

REL07-BP02 Obtain resources upon detection of impairment to a workload

Scale resources reactively when necessary if availability is impacted, to restore workload availability.

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

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

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

- Use scaling plans, which are the core component of AWS Auto Scaling, to configure a set of instructions for scaling your resources. If you work with AWS CloudFormation or add tags to AWS resources, you can set up scaling plans for different sets of resources, per application. AWS Auto Scaling provides recommendations for scaling strategies customized to each resource. After you create your scaling plan, AWS Auto Scaling combines dynamic scaling and predictive scaling methods together to support your scaling strategy.
  - AWS Auto Scaling: How Scaling Plans Work
- Amazon EC2 Auto Scaling helps you ensure that you have the correct number of Amazon EC2 instances available to handle the load for your application. You create collections of EC2 instances, called Auto Scaling groups. You can specify the minimum number of instances in each Auto Scaling group, and Amazon EC2 Auto Scaling ensures that your group never goes below this size. You can specify the maximum number of instances in each Auto Scaling group, and Amazon EC2 Auto Scaling ensures that your group never goes above this size.
  - What Is Amazon EC2 Auto Scaling?
- Amazon DynamoDB auto scaling uses the AWS Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your behalf, in response to actual traffic patterns. This enables a table or a global secondary index to increase its provisioned read and write capacity to handle sudden increases in traffic, without throttling.
  - Managing Throughput Capacity Automatically with DynamoDB Auto Scaling

Resources

Related documents:

- APN Partner: partners that can help you create automated compute solutions
- AWS Auto Scaling: How Scaling Plans Work
- AWS Marketplace: products that can be used with auto scaling
- Managing Throughput Capacity Automatically with DynamoDB Auto Scaling
- What Is Amazon EC2 Auto Scaling?

REL07-BP03 Obtain resources upon detection that more resources are needed for a workload

Scale resources proactively to meet demand and avoid availability impact.

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

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

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

$L = \lambda W$

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

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

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

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Obtain resources upon detection that more resources are needed for a workload. Scale resources proactively to meet demand and avoid availability impact.
  - Calculate how many compute resources you will need (compute concurrency) to handle a given request rate.
    - Telling Stories About Little's Law
  - When you have a historical pattern for usage, set up scheduled scaling for Amazon EC2 auto scaling.
    - Scheduled Scaling for Amazon EC2 Auto Scaling
  - Use AWS predictive scaling.
    - Predictive Scaling for EC2, Powered by Machine Learning

Resources

Related documents:

- AWS Auto Scaling: How Scaling Plans Work
- AWS Marketplace: products that can be used with auto scaling
- Managing Throughput Capacity Automatically with DynamoDB Auto Scaling
- Predictive Scaling for EC2, Powered by Machine Learning
- Scheduled Scaling for Amazon EC2 Auto Scaling
- Telling Stories About Little's Law
- What Is Amazon EC2 Auto Scaling?

REL07-BP04 Load test your workload

Adopt a load testing methodology to measure if scaling activity meets workload requirements.

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

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

Common anti-patterns:

- Performing load testing on deployments that are not the same configuration as your production.
- Performing load testing only on individual pieces of your workload, and not on the entire workload.
- Performing load testing with a subset of requests and not a representative set of actual requests.

- Performing load testing to a small safety factor above expected load.

Benefits of establishing this best practice: You know what components in your architecture fail under load and be able to identify what metrics to watch to indicate that you are approaching that load in time to address the problem, preventing the impact of that failure.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Perform load testing to identify which aspect of your workload indicates that you must add or remove capacity. Load testing should have representative traffic similar to what you receive in production. Increase the load while watching the metrics you have instrumented to determine which metric indicates when you must add or remove resources.
  - Distributed Load Testing on AWS: simulate thousands of connected users
    - Identify the mix of requests. You may have varied mixes of requests, so you should look at various time frames when identifying the mix of traffic.
    - Implement a load driver. You can use custom code, open source, or commercial software to implement a load driver.
    - Load test initially using small capacity. You see some immediate effects by driving load onto a lesser capacity, possibly as small as one instance or container.
    - Load test against larger capacity. The effects will be different on a distributed load, so you must test against as close to a product environment as possible.

Resources

Related documents:

- Distributed Load Testing on AWS: simulate thousands of connected users

Implement change

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

Best practices

REL08-BP01 Use runbooks for standard activities such as deployment

Runbooks are the predefined procedures to achieve specific outcomes. Use runbooks to perform standard activities, whether done manually or automatically. Examples include deploying a workload, patching a workload, or making DNS modifications.

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

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

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

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

Common anti-patterns:

- Performing unplanned changes to configuration in production.
- Skipping steps in your plan to deploy faster, resulting in a failed deployment.
- Making changes without testing the reversal of the change.

Benefits of establishing this best practice: Effective change planning increases your ability to successfully execute the change because you are aware of all the systems impacted. Validating your change in test environments increases your confidence.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Enable consistent and prompt responses to well understood events by documenting procedures in runbooks.
  - AWS Well-Architected Framework: Concepts: Runbook
- Use the principle of infrastructure as code to define your infrastructure. By using AWS CloudFormation (or a trusted third party) to define your infrastructure, you can use version control software to version and track changes.
  - Use AWS CloudFormation (or a trusted third-party provider) to define your infrastructure.
    - What is AWS CloudFormation?
  - Create templates that are singular and decoupled, using good software design principles.
    - Determine the permissions, templates, and responsible parties for implementation.
      - Controlling access with AWS Identity and Access Management
    - Use source control, like AWS CodeCommit or a trusted third-party tool, for version control.
      - What is AWS CodeCommit?

Resources

Related documents:

- APN Partner: partners that can help you create automated deployment solutions
- AWS Marketplace: products that can be used to automate your deployments
- AWS Well-Architected Framework: Concepts: Runbook
- What is AWS CloudFormation?
- What is AWS CodeCommit?

Related examples:

- Automating operations with Playbooks and Runbooks

REL08-BP02 Integrate functional testing as part of your deployment

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Integrate functional testing as part of your deployment. Functional tests are run as part of automated deployment. If success criteria are not met, the pipeline is halted or rolled back.
  - Invoke AWS CodeBuild during the 'Test Action' of your software release pipelines modeled in AWS CodePipeline. This capability enables you to easily run a variety of tests against your code, such as unit tests, static code analysis, and integration tests.
    - AWS CodePipeline Adds Support for Unit and Custom Integration Testing with AWS CodeBuild
  - Use AWS Marketplace solutions for executing automated tests as part of your software delivery pipeline.
    - Software test automation

Resources

Related documents:

- AWS CodePipeline Adds Support for Unit and Custom Integration Testing with AWS CodeBuild
- Software test automation
- What Is AWS CodePipeline?

REL08-BP03 Integrate resiliency testing as part of your deployment

Resiliency tests (using the principles of chaos engineering) are run as part of the automated deployment pipeline in a pre-production environment.

These tests are staged and run in the pipeline in a pre-production environment. They should also be run in production as part of game days.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Integrate resiliency testing as part of your deployment. Use Chaos Engineering, the discipline of experimenting on a workload to build confidence in the workload's capability to withstand turbulent conditions in production.
  - Resiliency tests inject faults or resource degradation to assess that your workload responds with its designed resilience.
    - Well-Architected lab: Level 300: Testing for Resiliency of EC2 RDS and S3

- These tests can be run regularly in pre-production environments in automated deployment pipelines.
- They should also be run in production, as part of scheduled game days.
- Using Chaos Engineering principles, propose hypotheses about how your workload will perform under various impairments, then test your hypotheses using resiliency testing.
  - Principles of Chaos Engineering

Resources

Related documents:

- Principles of Chaos Engineering
- What is AWS Fault Injection Simulator?

Related examples:

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

REL08-BP04 Deploy using immutable infrastructure

Immutable infrastructure is a model that mandates that no updates, security patches, or configuration changes happen in-place on production workloads. When a change is needed, the architecture is built onto new infrastructure and deployed into production.

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

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

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

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

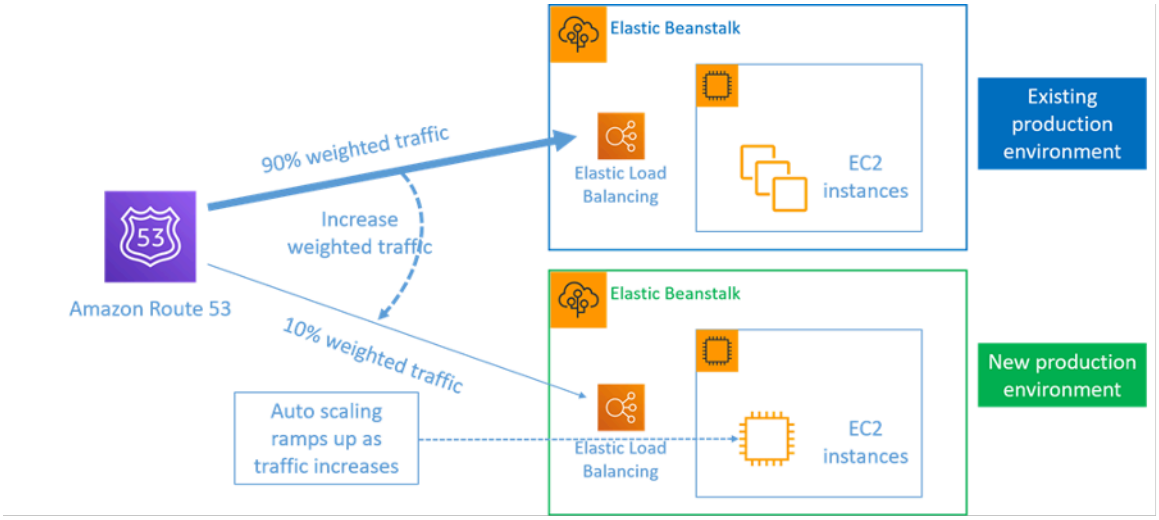


Figure 8: Blue/green deployment with AWS Elastic Beanstalk and Amazon Route 53

Benefits of immutable infrastructure:

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Deploy using immutable infrastructure. Immutable infrastructure is a model in which no updates, security patches, or configuration changes happen *in-place* on production systems. If any change is needed, a new version of the architecture is built and deployed into production.
  - Overview of a Blue/Green Deployment
  - Deploying Serverless Applications Gradually
  - Immutable Infrastructure: Reliability, consistency and confidence through immutability
  - CanaryRelease

Resources

Related documents:

- CanaryRelease
- Deploying Serverless Applications Gradually
- Immutable Infrastructure: Reliability, consistency and confidence through immutability
- Overview of a Blue/Green Deployment
- The Amazon Builders' Library: Ensuring rollback safety during deployments

REL08-BP05 Deploy changes with automation

Deployments and patching are automated to eliminate negative impact.

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

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

Common anti-patterns:

- Manually performing changes.
- Skipping steps in your automation through emergency work flows.
- Not following your plans.

Benefits of establishing this best practice: Using automation to deploy all changes removes the potential for introduction of human error and enables the ability to test before changing production to ensure that your plans are complete.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Automate your deployment pipeline. Deployment pipelines allow you to invoke automated testing and detection of anomalies, and either halt the pipeline at a certain step before production deployment, or automatically roll back a change.
  - The Amazon Builders' Library: Ensuring rollback safety during deployments
  - The Amazon Builders' Library: Going faster with continuous delivery
    - Use AWS CodePipeline (or a trusted third-party product) to define and run your pipelines.
      - Configure the pipeline to start when a change is committed to your code repository.
        - What is AWS CodePipeline?
      - Use Amazon Simple Notification Service (Amazon SNS) and Amazon Simple Email Service (Amazon SES) to send notifications about problems in the pipeline or integrate with a team chat tool, like Amazon Chime.
        - What is Amazon Simple Notification Service?

- What is Amazon SES?
- What is Amazon Chime?
- Automate chat messages with webhooks.

Resources

Related documents:

- APN Partner: partners that can help you create automated deployment solutions
- AWS Marketplace: products that can be used to automate your deployments
- Automate chat messages with webhooks.
- The Amazon Builders' Library: Ensuring rollback safety during deployments
- The Amazon Builders' Library: Going faster with continuous delivery
- What Is AWS CodePipeline?
- What Is CodeDeploy?
- AWS Systems Manager Patch Manager
- What is Amazon SES?
- What is Amazon Simple Notification Service?

Related videos:

- AWS Summit 2019: CI/CD on AWS

Failure management

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

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

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

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

Topics

Back up data

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

Best practices

REL09-BP01 Identify and back up all data that needs to be backed up, or reproduce the data from sources

All AWS data stores offer backup capabilities. Services such as Amazon RDS and Amazon DynamoDB additionally support automated backup that enables point-in-time recovery (PITR), which allows you to

restore a backup to any time up to five minutes or less before the current time. Many AWS services offer the ability to copy backups to another AWS Region. AWS Backup is a tool that gives you the ability to centralize and automate data protection across AWS services.

Amazon S3 can be used as a backup destination for self-managed and AWS-managed data sources. AWS services such as Amazon EBS, Amazon RDS, and Amazon DynamoDB have built in capabilities to create backups. Third-party backup software can also be used.

On-premises data can be backed up to the AWS Cloud using AWS Storage Gateway or AWS DataSync. Amazon S3 buckets can be used to store this data on AWS. Amazon S3 offers multiple storage tiers such as Amazon S3 Glacier or S3 Glacier Deep Archive to reduce cost of data storage.

You might be able to meet data recovery needs by reproducing the data from other sources. For example, Amazon Elasticache replica nodes or RDS read replicas could be used to reproduce data if the primary is lost. In cases where sources like this can be used to meet your Recovery Point Objective (RPO) and Recovery Time Objective (RTO), you might not require a backup. Another example, if working with Amazon EMR, it might not be necessary to backup your HDFS data store, as long as you can reproduce the data into EMR from S3.

When selecting a backup strategy, consider the time it takes to recover data. The time needed to recover data depends on the type of backup (in the case of a backup strategy), or the complexity of the data reproduction mechanism. This time should fall within the RTO for the workload.

Desired Outcome:

Data sources have been identified and classified based on criticality. Then, establish a strategy for data recovery based on the RPO. This strategy involves either backing up these data sources, or having the ability to reproduce data from other sources. In the case of data loss, the strategy implemented enables recovery or reproduction of data within the defined RPO and RTO.

Cloud Maturity Phase: Foundational

Common anti-patterns:

- Not aware of all data sources for the workload and their criticality.
- Not taking backups of critical data sources.
- Taking backups of only some data sources without using criticality as a criterion.
- No defined RPO, or backup frequency cannot meet RPO.
- Not evaluating if a backup is necessary or if data can be reproduced from other sources.

Benefits of establishing this best practice: Identifying the places where backups are necessary and implementing a mechanism to create backups, or being able to reproduce the data from an external source improves the ability to restore and recover data during an outage.

Level of risk exposed if this best practice is not established: High

Implementation guidance

Understand and use the backup capabilities of the AWS services and resources used by the workload. Most AWS services provides capabilities to back up workload data.

Implementation Steps:

1. **Identify all data sources for the workload**. Data can be stored on a number of resources such as databases, volumes, filesystems, logging systems, and object storage. Refer to the **Resources** section to find **Related documents** on different AWS services where data is stored, and the backup capability these services provide.

2. **Classify data sources based on criticality**. Different data sets will have different levels of criticality for a workload, and therefore different requirements for resiliency. For example, some data might be critical and require a RPO near zero, while other data might be less critical and can tolerate a higher RPO and some data loss. Similarly, different data sets might have different RTO requirements as well.
3. **Use AWS or third-party services to create backups of the data**. AWS Backup is a managed service that enables creating backups of various data sources on AWS. Most of these services also have native capabilities to create backups. The AWS Marketplace has many solutions that provide these capabilites as well. Refer to the **Resources** listed below for information on how to create backups of data from various AWS services.
4. **For data that is not backed up, establish a data reproduction mechanism**. You might choose not to backup data that can be reproduced from other sources for various reasons. There might be a situation where it is cheaper to reproduce data from sources when needed rather than creating a backup as there may be a cost associated with storing backups. Another example is where restoring from a backup takes longer than reproducing the data from sources, resulting in a breach in RTO. In such situations, consider tradeoffs and establish a well-defined process for how data can be reproduced from these sources when data recovery is necessary. For example, if you have loaded data from Amazon S3 to a data warehouse (like Amazon Redshift), or MapReduce cluster (like Amazon EMR) to do analysis on that data, this may be an example of data that can be reproduced from other sources. As long as the results of these analyses are either stored somewhere or reproducible, you would not suffer a data loss from a failure in the data warehouse or MapReduce cluster. Other examples that can be reproduced from sources include caches (like Amazon ElastiCache) or RDS read replicas.
5. **Establish a cadence for backing up data**. Creating backups of data sources is a periodic process and the frequency should depend on the RPO.

Level of effort for the Implementation Plan: Moderate

Resources

Related Best Practices:

REL13-BP01 Define recovery objectives for downtime and data loss (p. 100)

REL13-BP02 Use defined recovery strategies to meet the recovery objectives (p. 105)

Related documents:

- What Is AWS Backup?
- What is AWS DataSync?
- What is Volume Gateway?
- APN Partner: partners that can help with backup
- AWS Marketplace: products that can be used for backup
- Amazon EBS Snapshots
- Backing Up Amazon EFS
- Backing up Amazon FSx for Windows File Server
- Backup and Restore for ElastiCache for Redis
- Creating a DB Cluster Snapshot in Neptune
- Creating a DB Snapshot
- Creating an EventBridge Rule That Triggers on a Schedule
- Cross-Region Replication with Amazon S3
- EFS-to-EFS AWS Backup

- Exporting Log Data to Amazon S3
- Object lifecycle management
- On-Demand Backup and Restore for DynamoDB
- Point-in-time recovery for DynamoDB
- Working with Amazon OpenSearch Service Index Snapshots

Related videos:

- AWS re:Invent 2021 - Backup, disaster recovery, and ransomware protection with AWS
- AWS Backup Demo: Cross-Account and Cross-Region Backup
- AWS re:Invent 2019: Deep dive on AWS Backup, ft. Rackspace (STG341)

Related examples:

- Well-Architected lab: Implementing Bi-Directional Cross-Region Replication (CRR) for Amazon S3
- Well-Architected lab: Testing Backup and Restore of Data
- Well-Architected lab: Backup and Restore with Failback for Analytics Workload
- Well-Architected lab: Disaster Recovery - Backup and Restore

REL09-BP02 Secure and encrypt backups

Control and detect access to backups using authentication and authorization, such as AWS IAM. Prevent and detect if data integrity of backups is compromised using encryption.

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

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

Common anti-patterns:

- Having the same access to the backups and restoration automation as you do to the data.
- Not encrypting your backups.

Benefits of establishing this best practice: Securing your backups prevents tampering with the data, and encryption of the data prevents access to that data if it is accidentally exposed.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Use encryption on each of your data stores. If your source data is encrypted, then the backup will also be encrypted.
  - Enable encryption in RDS. You can configure encryption at rest using AWS Key Management Service when you create an RDS instance.
    - Encrypting Amazon RDS Resources

- Enable encryption on EBS volumes. You can configure default encryption or specify a unique key upon volume creation.
  - Amazon EBS Encryption
- Use the required Amazon DynamoDB encryption. DynamoDB encrypts all data at rest. You can either use an AWS owned AWS KMS key or an AWS managed KMS key, specifying a key that is stored in your account.
  - DynamoDB Encryption at Rest
  - Managing Encrypted Tables
- Encrypt your data stored in Amazon EFS. Configure the encryption when you create your file system.
  - Encrypting Data and Metadata in EFS
- Configure the encryption in the source and destination Regions. You can configure encryption at rest in Amazon S3 using keys stored in KMS, but the keys are Region-specific. You can specify the destination keys when you configure the replication.
  - CRR Additional Configuration: Replicating Objects Created with Server-Side Encryption (SSE) Using Encryption Keys stored in AWS KMS
- Implement least privilege permissions to access your backups. Follow best practices to limit the access to the backups, snapshots, and replicas in accordance with security best practices.
  - Security Pillar: AWS Well-Architected

Resources

Related documents:

- AWS Marketplace: products that can be used for backup
- Amazon EBS Encryption
- Amazon S3: Protecting Data Using Encryption
- CRR Additional Configuration: Replicating Objects Created with Server-Side Encryption (SSE) Using Encryption Keys stored in AWS KMS
- DynamoDB Encryption at Rest
- Encrypting Amazon RDS Resources
- Encrypting Data and Metadata in EFS
- Encryption for Backups in AWS
- Managing Encrypted Tables
- Security Pillar: AWS Well-Architected

Related examples:

- Well-Architected lab: Implementing Bi-Directional Cross-Region Replication (CRR) for Amazon S3

REL09-BP03 Perform data backup automatically

Configure backups to be taken automatically based on a periodic schedule informed by the Recovery Point Objective (RPO), or by changes in the dataset. Critical datasets with low data loss requirements need to be backed up automatically on a frequent basis, whereas less critical data where some loss is acceptable can be backed up less frequently.

AWS Backup can be used to create automated data backups of various AWS data sources. Amazon RDS instances can be backed up almost continuously every five minutes and Amazon S3 objects can be backed up almost continuously every fifteen minutes, providing for point-in-time recovery (PITR)

to a specific point in time within the backup history. For other AWS data sources, such as Amazon EBS volumes, Amazon DynamoDB tables, or Amazon FSx file systems, AWS Backup can run automated backup as frequently as every hour. These services also offer native backup capabilities. AWS services that offer automated backup with point-in-time recovery include Amazon DynamoDB, Amazon RDS, and Amazon Keyspaces (for Apache Cassandra) – these can be restored to a specific point in time within the backup history. Most other AWS data storage services offer the ability to schedule periodic backups, as frequently as every hour.

Amazon RDS and Amazon DynamoDB offer continuous backup with point-in-time recovery. Amazon S3 versioning, once enabled, is automatic. Amazon Data Lifecycle Manager can be used to automate the creation, copy and deletion of Amazon EBS snapshots. It can also automate the creation, copy, deprecation and deregistration of Amazon EBS-backed Amazon Machine Images (AMIs) and their underlying Amazon EBS snapshots.

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

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

Desired Outcome:

An automated process that creates backups of data sources at an established cadence.

Common anti-patterns:

- Performing backups manually.
- Using resources that have backup capability, but not including the backup in your automation.

Benefits of establishing this best practice: Automating backups ensures that they are taken regularly based on your RPO, and alerts you if they are not taken.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

1. **Identify data sources** that are currently being backed up manually. Refer to REL09-BP01 Identify and back up all data that needs to be backed up, or reproduce the data from sources (p. 64) for guidance on this.
2. **Determine the RPO** for the workload. Refer to REL13-BP01 Define recovery objectives for downtime and data loss (p. 100) for guidance on this.
3. **Use an automated backup solution or managed service**. AWS Backup is a fully-managed service that makes it easy to centralize and automate data protection across AWS services, in the cloud, and on premises. Backup plans are a feature of AWS Backup that enables the creation of rules which define the resources to backup, and the frequency at which these backups should be created. This frequency should be informed by the RPO established in Step 2. This WA Lab provides hands-on guidance on how to create automated backups using AWS Backup. Native backup capabilities are offered by most AWS services that store data. For example, RDS can be leveraged for automated backups with point-in-time recovery (PITR).
4. **For data sources not supported** by an automated backup solution or managed service such as on-premises data sources or message queues, consider using a trusted third-party solution to create automated backups. Alternatively, you can create automation to do this using the AWS CLI or SDKs. You can use AWS Lambda Functions or AWS Step Functions to define the logic involved in creating a data backup, and use Amazon EventBridge to execute it at a frequency based on your RPO (as established in Step 2).

Level of effort for the Implementation Plan: Low

Resources

Related documents:

- APN Partner: partners that can help with backup
- AWS Marketplace: products that can be used for backup
- Creating an EventBridge Rule That Triggers on a Schedule
- What Is AWS Backup?
- What Is AWS Step Functions?

Related videos:

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

Related examples:

- Well-Architected lab: Testing Backup and Restore of Data

REL09-BP04 Perform periodic recovery of the data to verify backup integrity and processes

Validate that your backup process implementation meets your recovery time objectives (RTO) and recovery point objectives (RPO) by performing a recovery test.

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

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

Desired Outcome: Data from backups is periodically recovered using well-defined mechanisms to ensure that recovery is possible within the established recovery time objective (RTO) for the workload. Verify that restoration from a backup results in a resource that contains the original data without any of it being corrupted or inaccessible, and with data loss within the recovery point objective (RPO).

Common anti-patterns:

- Restoring a backup, but not querying or retrieving any data to ensure that the restoration is usable.
- Assuming that a backup exists.
- Assuming that the backup of a system is fully operational and that data can be recovered from it.
- Assuming that the time to restore or recover data from a backup falls within the RTO for the workload.
- Assuming that the data contained on the backup falls within the RPO for the workload
- Restoring ad hoc, without using a runbook, or outside of an established automated procedure.

Benefits of establishing this best practice: Testing the recovery of the backups ensures data can be restored when needed without having any worry that data might be missing or corrupted, that the restoration and recovery is possible within the RTO for the workload, and any data loss falls within the RPO for the workload.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

Testing backup and restore capability increases confidence in the ability to perform these actions during an outage. Periodically restore backups to a new location and run tests to verify the integrity of the data. Some common tests that should be performed are checking

if all the data is available, is not corrupted, is accessible, and any data loss falls within the RPO for the workload. Such tests can also help ascertain if recovery mechanisms are fast enough to accommodate the workload's RTO.

1. **Identify data sources** that are currently being backed up and where these backups are being stored. Refer to REL09-BP01 Identify and back up all data that needs to be backed up, or reproduce the data from sources (p. 64) for guidance on how to implement this.
2. **Establish criteria for data validation** for each data source. Different types of data will have different properties which might require different validation mechanisms. Consider how this data might be validated before you are confident to use it in production. Some common ways to validate data are using data and backup properties such as data type, format, checksum, size, or a combination of these with custom validation logic. For example, this might be a comparison of the checksum values between the restored resource and the data source at the time the backup was created.
3. **Establish RTO and RPO** for restoring the data based on data criticality. Refer to REL13-BP01 Define recovery objectives for downtime and data loss (p. 100) for guidance on how to implement this.
4. **Assess your recovery capability**. Review your backup and restore strategy to understand if it can meet your RTO and RPO, and adjust the strategy as necessary. Using AWS Resilience Hub, you can run an assessment of your workload. The assessment evaluates your application configuration against the resiliency policy and reports if your RTO and RPO targets can be met.
5. **Do a test restore** using currently established processes used in production for data restoration. These processes depend on how the original data source was backed up, the format and storage location of the backup itself, or if the data is reproduced from other sources. For example, if you are using a managed service such as AWS Backup, this might be as simple as restoring the backup into a new resource. If you used AWS Elastic Disaster Recovery you can launch a recovery drill.
6. **Validate data recovery** from the restored resource (from the previous step) based on criteria you previously established for data validation in step 2. Does the restored and recovered data contain the most recent record/item at the time of backup? Does this data fall within the RPO for the workload?
7. **Measure time required** for restore and recovery and compare it to RTO established earlier in step 3. Does this process fall within the RTO for the workload? For example, compare the timestamps from when the restoration process started and when the recovery validation completed to calculate how long this process takes. All AWS API calls are timestamped and this information is available in AWS CloudTrail. While this information can provide details on when the restore process started, the end timestamp for when the validation was completed should be recorded by your validation logic. If using an automated process, then services like Amazon DynamoDB can be used to store this information. Additionally, many AWS services provide an event history which provides timestamped information when certain actions occurred. Within AWS Backup, backup and restore actions are referred to as *Jobs*, and these Jobs contain timestamp information as part of its metadata which can be used to measure time required for restoration and recovery.
8. **Notify stakeholders** if data validation fails, or if the time required for restoration and recovery exceeds the established RTO for the workload. When implementing automation to do this, such as in this lab, services like Amazon Simple Notification Service (Amazon SNS) can be used to send push notifications such as email or SMS to stakeholders. These messages can also be published to messaging applications such as Amazon Chime, Slack, or Microsoft Teams or used to create tasks as OpsItems using AWS Systems Manager OpsCenter.
9. **Automate this process to run periodically**. For example, services like AWS Lambda or a State Machine in AWS Step Functions can be used to automate the restore and recovery processes, and Amazon EventBridge can be used to trigger this automation workflow periodically as shown in the architecture diagram below. Learn how to Automate data recovery validation with AWS Backup. Additionally, this

Well-Architected lab provides a hands-on experience on one way to do automation for several of the steps here.

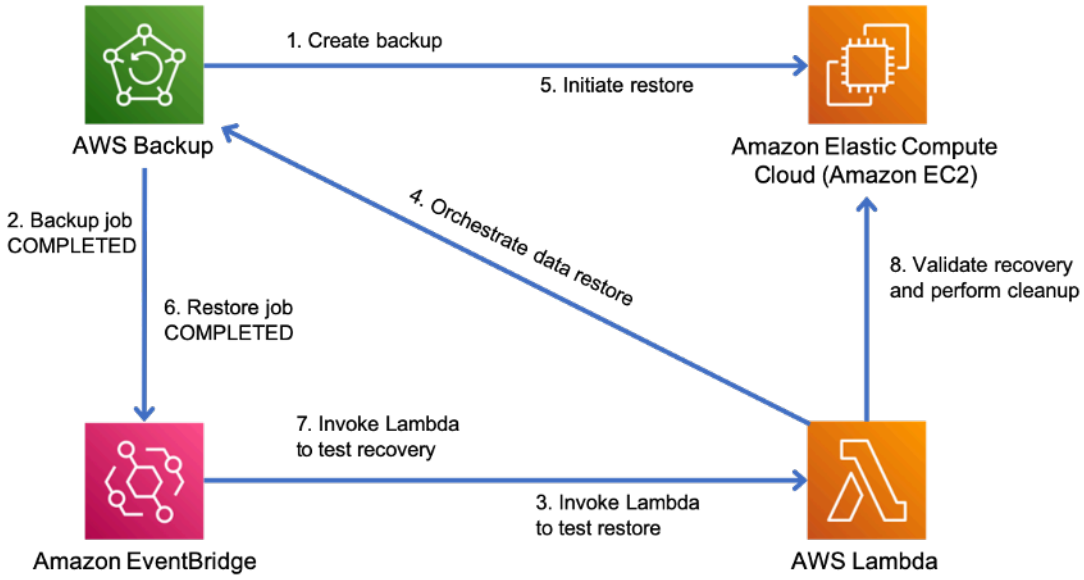


Figure 9. An automated backup and restore process

Level of effort for the Implementation Plan: Moderate to high depending on the complexity of the validation criteria.

Resources

Related documents:

- Automate data recovery validation with AWS Backup
- APN Partner: partners that can help with backup
- AWS Marketplace: products that can be used for backup
- Creating an EventBridge Rule That Triggers on a Schedule
- On-demand backup and restore for DynamoDB
- What Is AWS Backup?
- What Is AWS Step Functions?
- What is AWS Elastic Disaster Recovery
- AWS Elastic Disaster Recovery

Related examples:

- Well-Architected lab: Testing Backup and Restore of Data

Use fault isolation to protect your workload

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

Best practices

REL10-BP01 Deploy the workload to multiple locations

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

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

Availability Zones (AZs)

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

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

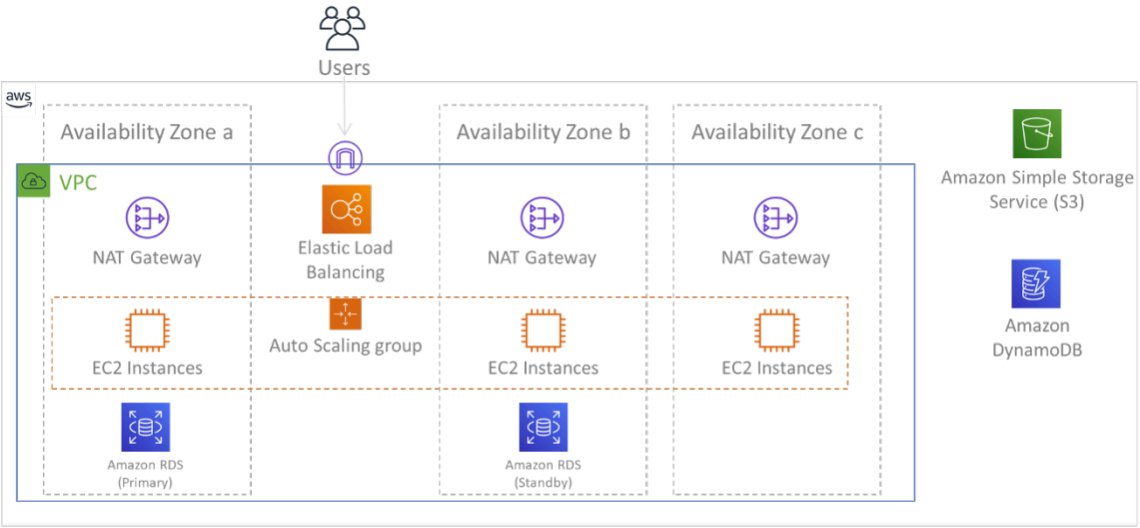


Figure 9: Multi-tier architecture deployed across three Availability Zones. Note that Amazon S3 and Amazon DynamoDB are always Multi-AZ automatically. The ELB also is deployed to all three zones.

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

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

AWS Local Zones

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

Amazon Global Edge Network

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

AWS Regions

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

A multi-Region approach is common for *disaster recovery* strategies to meet recovery objectives when one-off large-scale events occur. See Plan for Disaster Recovery (DR) for more information on these strategies. Here however, we focus instead on *availability*, which seeks to deliver a mean uptime objective over time. For high-availability objectives, a multi-region architecture will generally be designed to be active/active, where each service copy (in their respective regions) is active (serving requests).

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

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

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

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

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

On-premises data center

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Use multiple Availability Zones and AWS Regions. Distribute workload data and resources across multiple Availability Zones or, where necessary, across AWS Regions. These locations can be as diverse as required.
  - Regional services are inherently deployed across Availability Zones.
    - This includes Amazon S3, Amazon DynamoDB, and AWS Lambda (when not connected to a VPC)
  - Deploy your container, instance, and function-based workloads into multiple Availability Zones. Use multi-zone datastores, including caches. Use the features of EC2 Auto Scaling, ECS task placement, AWS Lambda function configuration when running in your VPC, and ElastiCache clusters.
    - Use subnets that are in separate Availability Zones when you deploy Auto Scaling groups.

        - Example: Distributing instances across Availability Zones
        - Amazon ECS task placement strategies
        - Configuring an AWS Lambda function to access resources in an Amazon VPC
        - Choosing Regions and Availability Zones
      - Use subnets in separate Availability Zones when you deploy Auto Scaling groups.
        - Example: Distributing instances across Availability Zones
      - Use ECS task placement parameters, specifying DB subnet groups.
        - Amazon ECS task placement strategies
      - Use subnets in multiple Availability Zones when you configure a function to run in your VPC.
        - Configuring an AWS Lambda function to access resources in an Amazon VPC
      - Use multiple Availability Zones with ElastiCache clusters.
        - Choosing Regions and Availability Zones
- If your workload must be deployed to multiple Regions, choose a multi-Region strategy. Most reliability needs can be met within a single AWS Region using a multi-Availability Zone strategy. Use a multi-Region strategy when necessary to meet your business needs.
  - AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)
    - Backup to another AWS Region can add another layer of assurance that data will be available when needed.
    - Some workloads have regulatory requirements that require use of a multi-Region strategy.
- Evaluate AWS Outposts for your workload. If your workload requires low latency to your on-premises data center or has local data processing requirements. Then run AWS infrastructure and services on premises using AWS Outposts
  - What is AWS Outposts?
- Determine if AWS Local Zones helps you provide service to your users. If you have low-latency requirements, see if AWS Local Zones is located near your users. If yes, then use it to deploy workloads closer to those users.
  - AWS Local Zones FAQ

Resources

Related documents:

- AWS Global Infrastructure
- AWS Local Zones FAQ
- Amazon ECS task placement strategies
- Choosing Regions and Availability Zones
- Example: Distributing instances across Availability Zones
- Global Tables: Multi-Region Replication with DynamoDB
- Using Amazon Aurora global databases
- Creating a Multi-Region Application with AWS Services blog series
- What is AWS Outposts?

Related videos:

- AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)
- AWS re:Invent 2019: Innovation and operation of the AWS global network infrastructure (NET339)

REL10-BP02 Select the appropriate locations for your multi-location deployment

Desired Outcome

For high availability, always (when possible) deploy your workload components to multiple Availability Zones (AZs), as shown in Figure 10. For workloads with extreme resilience requirements, carefully evaluate the options for a multi-Region architecture.

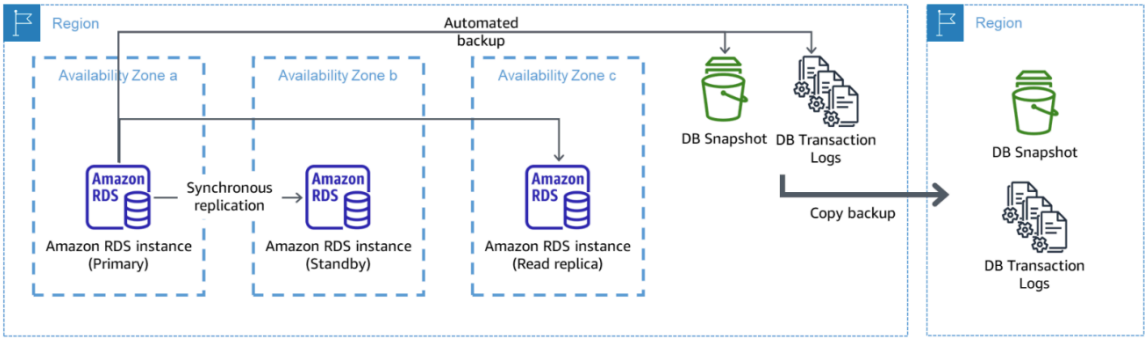


Figure 10: A resilient multi-AZ database deployment with backup to another AWS Region

Common anti-patterns

- Choosing to design a multi-Region architecture when a multi-AZ architecture would satisfy requirements.
- Not accounting for dependencies between application components if resilience and multi-location requirements differ between those components.

Benefits of establishing this best practice

For resilience, you should use an approach that builds layers of defense. One layer protects against smaller, more common, disruptions by building a highly available architecture using multiple AZs. Another layer of defense is meant to protect against rare events like widespread natural disasters and Region-level disruptions. This second layer involves architecting your application to span multiple AWS Regions.

- The difference between a 99.5% availability and 99.99% availability is over 3.5 hours per month. The expected availability of a workload can only reach “four nines” if it is in multiple AZs.
- By running your workload in multiple AZs, you can isolate faults in power, cooling, and networking, and most natural disasters like fire and flood.
- Implementing a multi-Region strategy for your workload helps protect it against widespread natural disasters that affect a large geographic region of a country, or technical failures of Region-wide scope. Be aware that implementing a multi-Region architecture can be significantly complex, and is usually not required for most workloads.

Level of risk exposed if this best practice is not established: High

Implementation guidance

For a disaster event based on disruption or partial loss of one Availability Zone, implementing a highly available workload in multiple Availability Zones within a single AWS Region helps mitigate against

natural and technical disasters. Each AWS Region is comprised of multiple Availability Zones, each isolated from faults in the other zones and separated by a meaningful distance. However, for a disaster event that includes the risk of losing multiple Availability Zone components, which are a significant distance away from each other, you should implement disaster recovery options to mitigate against failures of a Region-wide scope. For workloads that require extreme resilience (critical infrastructure, health-related applications, financial system infrastructure, etc.), a multi-Region strategy may be required.

Implementation Steps

1. Evaluate your workload and determine whether the resilience needs can be met by a multi-AZ approach (single AWS Region), or if they require a multi-Region approach. Implementing a multi-Region architecture to satisfy these requirements will introduce additional complexity, therefore carefully consider your use case and its requirements. Resilience requirements can almost always be met using a single AWS Region. Consider the following possible requirements when determining whether you need to use multiple Regions:
   a. **Disaster recovery (DR)**: For a disaster event based on disruption or partial loss of one Availability Zone, implementing a highly available workload in multiple Availability Zones within a single AWS Region helps mitigate against natural and technical disasters. For a disaster event that includes the risk of losing multiple Availability Zone components, which are a significant distance away from each other, you should implement disaster recovery across multiple Regions to mitigate against natural disasters or technical failures of a Region-wide scope.
   b. **High availability (HA)**: A multi-Region architecture (using multiple AZs in each Region) can be used to achieve greater then four 9's (> 99.99%) availability.
   c. **Stack localization**: When deploying a workload to a global audience, you can deploy localized stacks in different AWS Regions to serve audiences in those Regions. Localization can include language, currency, and types of data stored.
   d. **Proximity to users:** When deploying a workload to a global audience, you can reduce latency by deploying stacks in AWS Regions close to where the end users are.
   e. **Data residency**: Some workloads are subject to data residency requirements, where data from certain users must remain within a specific country's borders. Based on the regulation in question, you can choose to deploy an entire stack, or just the data, to the AWS Region within those borders.
2. Here are some examples of multi-AZ functionality provided by AWS services:
   a. To protect workloads using EC2 or ECS, deploy an Elastic Load Balancer in front of the compute resources. Elastic Load Balancing then provides the solution to detect instances in unhealthy zones and route traffic to the healthy ones.
      i. Getting started with Application Load Balancers
      ii. Getting started with Network Load Balancers
   b. In the case of EC2 instances running commercial off-the-shelf software that do not support load balancing, you can achieve a form of fault tolerance by implementing a multi-AZ disaster recovery methodology.
      i. the section called "REL13-BP02 Use defined recovery strategies to meet the recovery objectives" (p. 105)
   c. For Amazon ECS tasks, deploy your service evenly across three AZs to achieve a balance of availability and cost.
      i. Amazon ECS availability best practices | Containers
   d. For non-Aurora Amazon RDS, you can choose Multi-AZ as a configuration option. Upon failure of the primary database instance, Amazon RDS automatically promotes a standby database to receive traffic in another availability zone. Multi-Region read-replicas can also be created to improve resilience.
      i. Amazon RDS Multi AZ Deployments
      ii. Creating a read replica in a different AWS Region
3. Here are some examples of multi-Region functionality provided by AWS services:

a. For Amazon S3 workloads, where multi-AZ availability is provided automatically by the service, consider Multi-Region Access Points if a multi-Region deployment is needed.
   i. Multi-Region Access Points in Amazon S3
b. For DynamoDB tables, where multi-AZ availability is provided automatically by the service, you can easily convert existing tables to global tables to take advantage of multiple regions.
   i. Convert Your Single-Region Amazon DynamoDB Tables to Global Tables
c. If your workload is fronted by Application Load Balancers or Network Load Balancers, use AWS Global Accelerator to improve the availability of your application by directing traffic to multiple regions that contain healthy endpoints.
   i. Endpoints for standard accelerators in AWS Global Accelerator - AWS Global Accelerator (amazon.com)
d. For applications that leverage AWS EventBridge, consider cross-Region buses to forward events to other Regions you select.
   i. Sending and receiving Amazon EventBridge events between AWS Regions
e. For Amazon Aurora databases, consider Aurora global databases, which span multiple AWS regions. Existing clusters can be modified to add new Regions as well.
   i. Getting started with Amazon Aurora global databases
f. If your workload includes AWS Key Management Service (AWS KMS) encryption keys, consider whether multi-Region keys are appropriate for your application.
   i. Multi-Region keys in AWS KMS
g. For other AWS service features, see this blog series on Creating a Multi-Region Application with AWS Services series

Level of effort for the Implementation Plan: Moderate to High

Resources

Related Documents:

- Creating a Multi-Region Application with AWS Services series
- Disaster Recovery (DR) Architecture on AWS, Part IV: Multi-site Active/Active
- AWS Global Infrastructure
- AWS Local Zones FAQ
- Disaster Recovery (DR) Architecture on AWS, Part I: Strategies for Recovery in the Cloud
- Disaster recovery is different in the cloud
- Global Tables: Multi-Region Replication with DynamoDB

Related Videos:

- AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)
- Auth0: Multi-Region High-Availability Architecture that Scales to 1.5B+ Logins a Month with automated failover

Related Examples:

- Disaster Recovery (DR) Architecture on AWS, Part I: Strategies for Recovery in the Cloud
- DTCC achieves resilience well beyond what they can do on premises
- Expedia Group uses a multi-Region, multi-Availability Zone architecture with a proprietary DNS service to add resilience to the applications

- Uber: Disaster Recovery for Multi-Region Kafka
- Netflix: Active-Active for Multi-Regional Resilience
- How we build Data Residency for Atlassian Cloud
- Intuit TurboTax runs across two Regions

REL10-BP03 Automate recovery for components constrained to a single location

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

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

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

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Implement self-healing. Deploy your instances or containers using automatic scaling when possible. If you cannot use automatic scaling, use automatic recovery for EC2 instances or implement self-healing automation based on Amazon EC2 or ECS container lifecycle events.
  - Use Auto Scaling groups for instances and container workloads that have no requirements for a single instance IP address, private IP address, Elastic IP address, and instance metadata.
    - What Is EC2 Auto Scaling?
    - Service automatic scaling
      - The launch configuration user data can be used to implement automation that can self-heal most workloads.
  - Use automatic recovery of EC2 instances for workloads that require a single instance ID address, private IP address, Elastic IP address, and instance metadata.
    - Recover your instance.
      - Automatic Recovery will send recovery status alerts to a SNS topic as the instance failure is detected.
  - Use EC2 instance lifecycle events or ECS events to automate self-healing where automatic scaling or EC2 recovery cannot be used.
    - EC2 Auto Scaling lifecycle hooks
    - Amazon ECS events
      - Use the events to invoke automation that will heal your component according to the process logic you require.

Resources

Related documents:

- Amazon ECS events
- EC2 Auto Scaling lifecycle hooks
- Recover your instance.
- Service automatic scaling
- What Is EC2 Auto Scaling?

REL10-BP04 Use bulkhead architectures to limit scope of impact

Like the bulkheads on a ship, this pattern ensures that a failure is contained to a small subset of requests or clients so that the number of impaired requests is limited, and most can continue without error. Bulkheads for data are often called partitions, while bulkheads for services are known as cells.

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

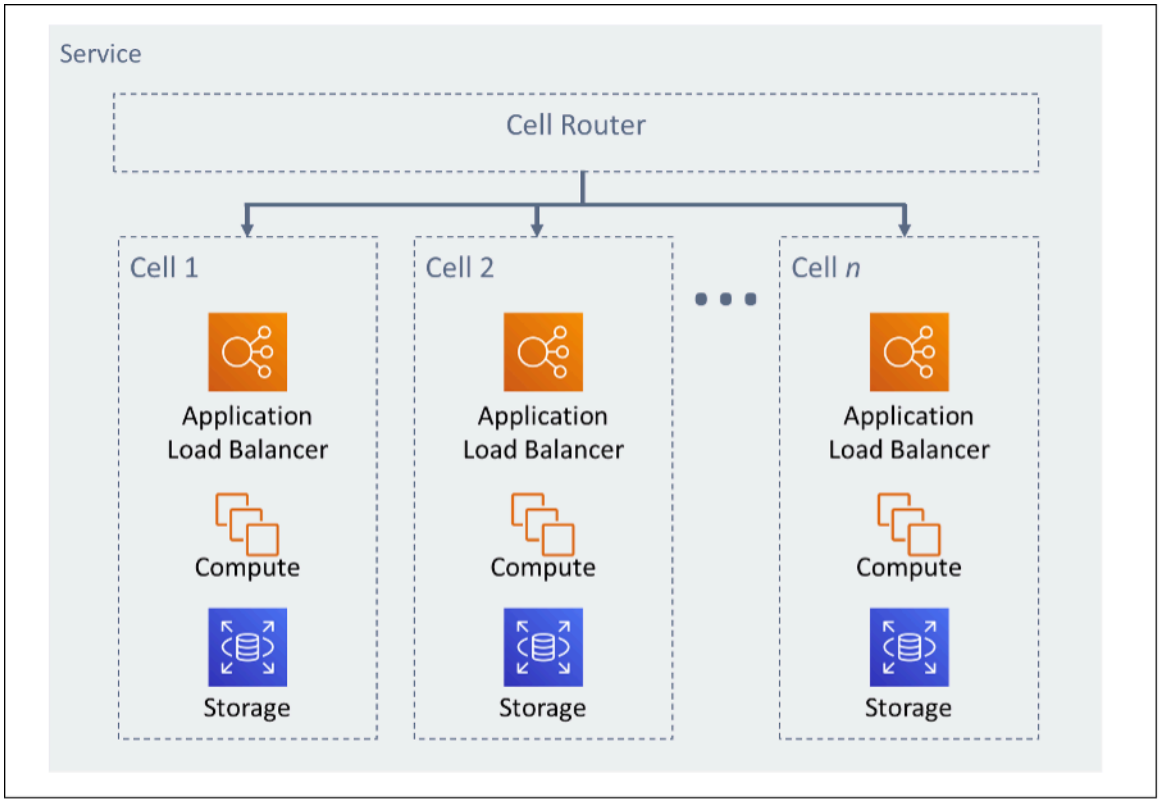


Figure 11: Cell-based architecture

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

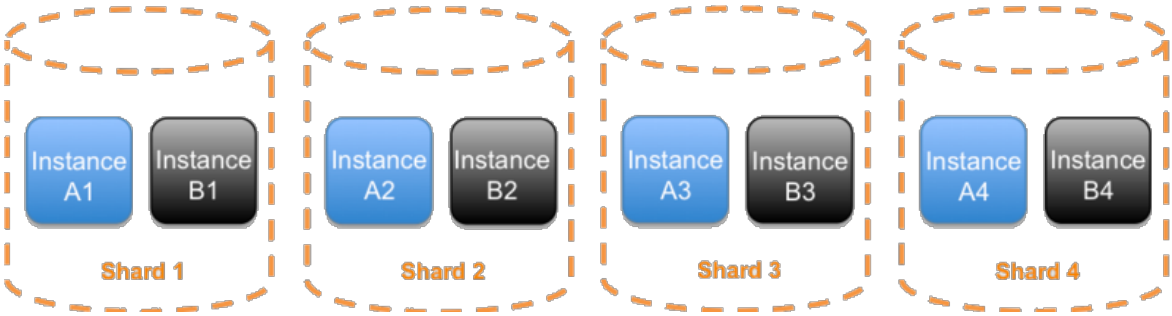


Figure 12: Service divided into four traditional shards of two cells each

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

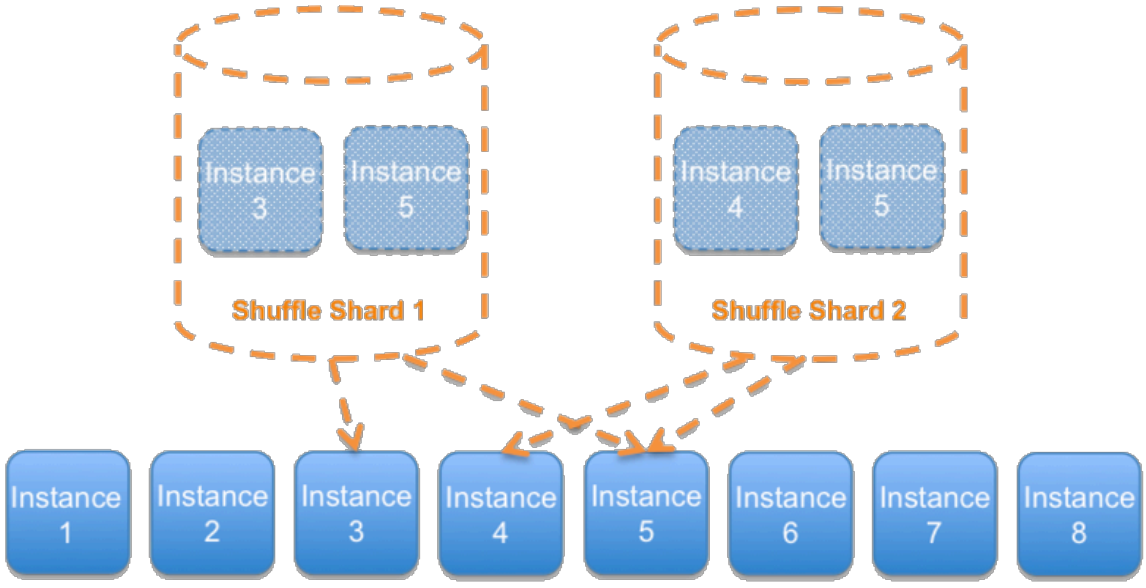


Figure 13: Service divided into 28 shuffle shards (virtual shards) of two cells each (only two shuffle shards out of the 28 possible are shown)

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Use bulkhead architectures. Like the bulkheads on a ship, this pattern ensures that a failure is contained to a small subset of requests or users so that the number of impaired requests is limited,

and most can continue without error. Bulkheads for data are often called partitions, while bulkheads for services are known as cells.

- Well-Architected lab: Fault isolation with shuffle sharding
- Shuffle-sharding: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)
- AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338)

- Evaluate cell-based architecture for your workload. In a cell-based architecture, each cell is a complete, independent instance of the service and has a fixed maximum size. As load increases, workloads grow by adding more cells. A partition key is used on incoming traffic to determine which cell will process the request. Any failure is contained to the single cell it occurs in, so that the number of impaired requests is limited as other cells continue without error. It is important to identify the proper partition key to minimize cross-cell interactions and avoid the need to involve complex mapping services in each request. Services that require complex mapping end up merely shifting the problem to the mapping services, while services that require cross-cell interactions reduce the autonomy of cells (and thus the assumed availability improvements of doing so).
  - In his AWS blog post, Colm MacCarthaigh explains how Amazon Route 53 uses the concept of shuffle sharding to isolate customer requests into shards
    - Shuffle Sharding: Massive and Magical Fault Isolation

Resources

Related documents:

- Shuffle Sharding: Massive and Magical Fault Isolation
- The Amazon Builders' Library: Workload isolation using shuffle-sharding

Related videos:

- AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338)
- Shuffle-sharding: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)

Related examples:

- Well-Architected lab: Fault isolation with shuffle sharding

Design your workload to withstand component failures

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

Best practices

- REL11-BP01 Monitor all components of the workload to detect failures (p. 84)
- REL11-BP02 Fail over to healthy resources (p. 85)
- REL11-BP03 Automate healing on all layers (p. 87)
- REL11-BP04 Rely on the data plane and not the control plane during recovery (p. 89)
- REL11-BP05 Use static stability to prevent bimodal behavior (p. 90)
- REL11-BP06 Send notifications when events impact availability (p. 92)

REL11-BP01 Monitor all components of the workload to detect failures

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

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

Common anti-patterns:

- No alarms have been configured, so outages occur without notification.
- Alarms exist, but at thresholds that don't provide adequate time to react.
- Metrics are not collected often enough to meet the recovery time objective (RTO).
- Only the customer facing tier of the workload is actively monitored.
- Only collecting technical metrics, no business function metrics.
- No metrics measuring the user experience of the workload.

Benefits of establishing this best practice: Having appropriate monitoring at all layers enables you to reduce recovery time by reducing time to detection.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Determine the collection interval for your components based on your recovery goals.
  - Your monitoring interval is dependent on how quickly you must recover. Your recovery time is driven by the time it takes to recover, so you must determine the frequency of collection by accounting for this time and your recovery time objective (RTO).
- Configure detailed monitoring for components.
  - Determine if detailed monitoring for EC2 instances and Auto Scaling is necessary. Detailed monitoring provides 1-min interval metrics, and default monitoring provides 5-minute interval metrics.
    - Enable or Disable Detailed Monitoring for Your Instance
    - Monitoring Your Auto Scaling Groups and Instances Using Amazon CloudWatch
  - Determine if enhanced monitoring for RDS is necessary. Enhanced monitoring uses an agent on the RDS instances to get useful information about different process or threads on an RDS instance.
    - Enhanced Monitoring
- Create custom metrics to measure business key performance indicators (KPIs). Workloads implement key business functions. These functions should be used as KPIs that help identify when an indirect problem happens.
  - Publishing Custom Metrics
- Monitor the user experience for failures using user canaries. Synthetic transaction testing (also known as canary testing, but not to be confused with canary deployments) that can run and simulate customer behavior is among the most important testing processes. Run these tests constantly against your workload endpoints from diverse remote locations.
  - Amazon CloudWatch Synthetics enables you to create user canaries

- Create custom metrics that track the user's experience. If you can instrument the experience of the customer, you can determine when the consumer experience degrades.
  - Publishing Custom Metrics
- Set alarms to detect when any part of your workload is not working properly, and to indicate when to Auto Scale resources. Alarms can be visually displayed on dashboards, send alerts via Amazon SNS or email, and work with Auto Scaling to scale up or down the resources for a workload.
  - Using Amazon CloudWatch Alarms
- Create dashboards to visualize your metrics. Dashboards can be used to visually see trends, outliers, and other indicators of potential problems, or to provide an indication of problems you may want to investigate.
  - Using CloudWatch Dashboards

Resources

Related documents:

- Amazon CloudWatch Synthetics enables you to create user canaries
- Enable or Disable Detailed Monitoring for Your Instance
- Enhanced Monitoring
- Monitoring Your Auto Scaling Groups and Instances Using Amazon CloudWatch
- Publishing Custom Metrics
- Using Amazon CloudWatch Alarms
- Using CloudWatch Dashboards

Related examples:

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

REL11-BP02 Fail over to healthy resources

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

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

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

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Fail over to healthy resources. Ensure that if a resource failure occurs, that healthy resources can continue to serve requests. For location failures (such as Availability Zone or AWS Region) ensure you have systems in place to fail over to healthy resources in unimpaired locations.
  - If your workload is using AWS services, such as Amazon S3 or Amazon DynamoDB, then they are automatically deployed to multiple Availability Zones. In case of failure, the AWS control plane automatically routes traffic to healthy locations for you.
  - For Amazon RDS you must choose Multi-AZ as a configuration option, and then on failure AWS automatically directs traffic to the healthy instance.
    - High Availability (Multi-AZ) for Amazon RDS
  - For Amazon EC2 instances or Amazon ECS tasks, you choose which Availability Zones to deploy to. Elastic Load Balancing then provides the solution to detect instances in unhealthy zones and route traffic to the healthy ones. Elastic Load Balancing can even route traffic to components in your on-premises data center.
  - For multi-region approaches (which might also include on-premises data centers), ensure that data and resources from healthy locations can continue to serve requests
    - For example, cross-region read replicas enable you to deploy your data to multiple AWS Regions, but you still must promote the read replica to master and point your traffic at it in the event of a primary location failure.
      - Overview of Amazon RDS Read Replicas
    - Amazon Route 53 provides a way to define internet domains, and assign routing policies, which might include health checks, to ensure that traffic is routed to healthy Regions. Alternately, AWS Global Accelerator provides static IP addresses that act as a fixed entry point to your application, then routes to endpoints in AWS Regions of your choosing, using the AWS global network instead of the public internet for better performance and reliability.
      - Amazon Route 53: Choosing a Routing Policy
      - What Is AWS Global Accelerator?

Resources

Related documents:

- APN Partner: partners that can help with automation of your fault tolerance
- AWS Marketplace: products that can be used for fault tolerance

- AWS OpsWorks: Using Auto Healing to Replace Failed Instances
- Amazon Route 53: Choosing a Routing Policy
- High Availability (Multi-AZ) for Amazon RDS
- Overview of Amazon RDS Read Replicas
- Amazon ECS task placement strategies
- Creating Kubernetes Auto Scaling Groups for Multiple Availability Zones
- What is AWS Global Accelerator?

Related examples:

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

REL11-BP03 Automate healing on all layers

Upon detection of a failure, use automated capabilities to perform actions to remediate.

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

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

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

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

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

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

Common anti-patterns:

- Deploying applications in instances or containers individually.
- Deploying applications that cannot be deployed into multiple locations without using automatic recovery.

- Manually healing applications that automatic scaling and automatic recovery fail to heal.

Benefits of establishing this best practice: Automated healing, even if the workload can only deployed into one location at a time will reduce your mean time to recovery, and ensure availability of the workload.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Use Auto Scaling groups to deploy tiers in an workload. Auto scaling can perform self-healing on stateless applications, and add and remove capacity.
  - How AWS Auto Scaling Works
- Implement automatic recovery on EC2 instances that have applications deployed that cannot be deployed in multiple locations, and can tolerate rebooting upon failures. Automatic recovery can be used to replace failed hardware and restart the instance when the application is not capable of being deployed in multiple locations. The instance metadata and associated IP addresses are kept, as well as the Amazon EBS volumes and mount points to Elastic File Systems or File Systems for Lustre and Windows.
  - Amazon EC2 Automatic Recovery
  - Amazon Elastic Block Store (Amazon EBS)
  - Amazon Elastic File System (Amazon EFS)
  - What is Amazon FSx for Lustre?
  - What is Amazon FSx for Windows File Server?
    - Using AWS OpsWorks, you can configure Auto Healing of EC2 instances at the layer level
      - AWS OpsWorks: Using Auto Healing to Replace Failed Instances
- Implement automated recovery using AWS Step Functions and AWS Lambda when you cannot use automatic scaling or automatic recovery, or when automatic recovery fails. When you cannot use automatic scaling, and either cannot use automatic recovery or automatic recovery fails, you can automate the healing using AWS Step Functions and AWS Lambda.
  - What is AWS Step Functions?
  - What is AWS Lambda?
    - Amazon EventBridge can be used to monitor and filter for events such as CloudWatch Alarms or changes in state in other AWS services. Based on event information, it can then trigger AWS Lambda (or other targets) to run custom remediation logic on your workload.
      - What Is Amazon EventBridge?
      - Using Amazon CloudWatch Alarms

Resources

Related documents:

- APN Partner: partners that can help with automation of your fault tolerance
- AWS Marketplace: products that can be used for fault tolerance
- AWS OpsWorks: Using Auto Healing to Replace Failed Instances
- Amazon EC2 Automatic Recovery
- Amazon Elastic Block Store (Amazon EBS)
- Amazon Elastic File System (Amazon EFS)
- How AWS Auto Scaling Works

- Using Amazon CloudWatch Alarms
- What Is Amazon EventBridge?
- What is AWS Lambda?
- AWS Systems Manager Automation
- What is AWS Step Functions?
- What is Amazon FSx for Lustre?
- What is Amazon FSx for Windows File Server?

Related videos:

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

Related examples:

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

REL11-BP04 Rely on the data plane and not the control plane during recovery

The control plane is used to configure resources, and the data plane delivers services. Data planes typically have higher availability design goals than control planes and are usually less complex. When implementing recovery or mitigation responses to potentially resiliency-impacting events, using control plane operations can lower the overall resiliency of your architecture. For example, you can rely on the Amazon Route 53 data plane to reliably route DNS queries based on health checks, but updating Route 53 routing policies uses the control plane, so do not rely on it for recovery.

The Route 53 data planes answer DNS queries, and perform and evaluate health checks. They are globally distributed and designed for a 100% availability service level agreement (SLA). The Route 53 management APIs and consoles where you create, update, and delete Route 53 resources run on control planes that are designed to prioritize the strong consistency and durability that you need when managing DNS. To achieve this, the control planes are located in a single Region, US East (N. Virginia). While both systems are built to be very reliable, the control planes are not included in the SLA. There could be rare events in which the data plane's resilient design allows it to maintain availability while the control planes do not. For disaster recovery and failover mechanisms, use data plane functions to provide the best possible reliability.

For more information about data planes, control planes, and how AWS builds services to meet high availability targets, see the Static stability using Availability Zones paper and the Amazon Builders' Library.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Rely on the data plane and not the control plane when using Amazon Route 53 for disaster recovery. Route 53 Application Recovery Controller helps you manage and coordinate failover using readiness checks and routing controls. These features continually monitor your application's ability to recover from failures, and enables you to control your application recovery across multiple AWS Regions, Availability Zones, and on premises.
  - What is Route 53 Application Recovery Controller
  - Creating Disaster Recovery Mechanisms Using Amazon Route 53

  - Building highly resilient applications using Amazon Route 53 Application Recovery Controller, Part 1: Single-Region stack
  - Building highly resilient applications using Amazon Route 53 Application Recovery Controller, Part 2: Multi-Region stack
- Understand which operations are on the data plane and which are on the control plane.
  - Amazon Builders' Library: Avoiding overload in distributed systems by putting the smaller service in control
  - Amazon DynamoDB API (control plane and data plane)
  - AWS Lambda Executions (split into the control plane and the data plane)
  - AWS Lambda Executions (split into the control plane and the data plane)

Resources

Related documents:

- APN Partner: partners that can help with automation of your fault tolerance
- AWS Marketplace: products that can be used for fault tolerance
- Amazon Builders' Library: Avoiding overload in distributed systems by putting the smaller service in control
- Amazon DynamoDB API (control plane and data plane)
- AWS Lambda Executions (split into the control plane and the data plane)
- AWS Elemental MediaStore Data Plane
- Building highly resilient applications using Amazon Route 53 Application Recovery Controller, Part 1: Single-Region stack
- Building highly resilient applications using Amazon Route 53 Application Recovery Controller, Part 2: Multi-Region stack
- Creating Disaster Recovery Mechanisms Using Amazon Route 53
- What is Route 53 Application Recovery Controller

Related examples:

- Introducing Amazon Route 53 Application Recovery Controller

REL11-BP05 Use static stability to prevent bimodal behavior

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

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

Figure 14: Static stability of EC2 instances across Availability Zones

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

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

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

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Use static stability to prevent bimodal behavior. Bimodal behavior is when your workload exhibits different behavior under normal and failure modes, for example, relying on launching new instances if an Availability Zone fails.
  - Minimizing Dependencies in a Disaster Recovery Plan
  - The Amazon Builders' Library: Static stability using Availability Zones
  - Static stability in AWS: AWS re:Invent 2019: Introducing The Amazon Builders' Library (DOP328)
    - You should instead build systems that are statically stable and operate in only one mode. In this case, provision enough instances in each zone to handle workload load if one AZ were removed and then use Elastic Load Balancing or Amazon Route 53 health checks to shift load away from the impaired instances.
    - Another example of bimodal behavior is allowing clients to bypass your workload cache when failures occur. This might seem to be a solution to accommodate client needs, but should not be allowed since it significantly changes demands on your workload and is likely to result in failures.

Resources

Related documents:

- Minimizing Dependencies in a Disaster Recovery Plan
- The Amazon Builders' Library: Static stability using Availability Zones

Related videos:

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

REL11-BP06 Send notifications when events impact availability

Notifications are sent upon the detection of significant events, even if the issue caused by the event was automatically resolved.

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

Common anti-patterns:

- Sending alarms that no one acts upon.
- Performing auto healing automation, but not notifying that healing was needed.

Benefits of establishing this best practice: Notifications of recovery events will ensure that you don't ignore problems that occur infrequently.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Alarms on business Key Performance Indicators when they exceed a low threshold Having a low threshold alarm on your business KPIs help you know when your workload is unavailable or non-functional.
  - Creating a CloudWatch Alarm Based on a Static Threshold
- Alarm on events that invoke healing automation You can directly invoke an SNS API to send notifications with any automation that you create.
  - What is Amazon Simple Notification Service?

Resources

Related documents:

- Creating a CloudWatch Alarm Based on a Static Threshold
- What Is Amazon EventBridge?
- What is Amazon Simple Notification Service?

Test reliability

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

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

Best practices

REL12-BP01 Use playbooks to investigate failures

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

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

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

Common anti-patterns:

- Planning to deploy a workload without knowing the processes to diagnose issues or respond to incidents.
- Unplanned decisions about which systems to gather logs and metrics from when investigating an event.
- Not retaining metrics and events long enough to be able to retrieve the data.

Benefits of establishing this best practice: Capturing playbooks ensures that processes can be consistently followed. Codifying your playbooks limits the introduction of errors from manual activity. Automating playbooks shortens the time to respond to an event by eliminating the requirement for team member intervention or providing them additional information when their intervention begins.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Use playbooks to identify issues. Playbooks are documented processes to investigate issues. Enable consistent and prompt responses to failure scenarios by documenting processes in playbooks.

Playbooks must contain the information and guidance necessary for an adequately skilled person to gather applicable information, identify potential sources of failure, isolate faults, and determine contributing factors (perform post-incident analysis).

- Implement playbooks as code. Perform your operations as code by scripting your playbooks to ensure consistency and limit reduce errors caused by manual processes. Playbooks can be composed of multiple scripts representing the different steps that might be necessary to identify the contributing factors to an issue. Runbook activities can be triggered or performed as part of playbook activities, or may prompt for execution of a playbook in response to identified events.
  - Automate your operational playbooks with AWS Systems Manager
  - AWS Systems Manager Run Command
  - AWS Systems Manager Automation
  - What is AWS Lambda?
  - What Is Amazon EventBridge?
  - Using Amazon CloudWatch Alarms

Resources

Related documents:

- AWS Systems Manager Automation
- AWS Systems Manager Run Command
- Automate your operational playbooks with AWS Systems Manager
- Using Amazon CloudWatch Alarms
- Using Canaries (Amazon CloudWatch Synthetics)
- What Is Amazon EventBridge?
- What is AWS Lambda?

Related examples:

- Automating operations with Playbooks and Runbooks

REL12-BP02 Perform post-incident analysis

Review customer-impacting events, and identify the contributing factors and preventative action items. Use this information to develop mitigations to limit or prevent recurrence. Develop procedures for prompt and effective responses. Communicate contributing factors and corrective actions as appropriate, tailored to target audiences. Have a method to communicate these causes to others as needed.

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

Common anti-patterns:

- Finding contributing factors, but not continuing to look deeper for other potential problems and approaches to mitigate.
- Only identifying human error causes, and not providing any training or automation that could prevent human errors.

Benefits of establishing this best practice: Conducting post-incident analysis and sharing the results enables other workloads to mitigate the risk if they have implemented the same contributing factors, and enables them to implement the mitigation or automated recovery before an incident occurs.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Establish a standard for your post-incident analysis. Good post-incident analysis provides opportunities to propose common solutions for problems with architecture patterns that are used in other places in your systems.
  - Ensure that the contributing factors are honest and blame free.
  - If you do not document your problems, you cannot correct them.
    - Ensure post-incident analysis is blame free so you can be dispassionate about the proposed corrective actions and promote honest self-assessment and collaboration on your application teams.
- Use a process to determine contributing factors. Have a process to identify and document the contributing factors of an event so that you can develop mitigations to limit or prevent recurrence and you can develop procedures for prompt and effective responses. Communicate contributing factors as appropriate, tailored to target audiences.
  - What is log analytics?

Resources

Related documents:

- What is log analytics?
- Why you should develop a correction of error (COE)

REL12-BP03 Test functional requirements

Use techniques such as unit tests and integration tests that validate required functionality.

You achieve the best outcomes when these tests are run automatically as part of build and deployment actions. For instance, using AWS CodePipeline, developers commit changes to a source repository where CodePipeline automatically detects the changes. Those changes are built, and tests are run. After the tests are complete, the built code is deployed to staging servers for testing. From the staging server, CodePipeline runs more tests, such as integration or load tests. Upon the successful completion of those tests, CodePipeline deploys the tested and approved code to production instances.

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Test functional requirements. These include unit tests and integration tests that validate required functionality.
  - Use CodePipeline with AWS CodeBuild to test code and run builds
  - AWS CodePipeline Adds Support for Unit and Custom Integration Testing with AWS CodeBuild
  - Continuous Delivery and Continuous Integration
  - Using Canaries (Amazon CloudWatch Synthetics)

- Software test automation

Resources

Related documents:

- APN Partner: partners that can help with implementation of a continuous integration pipeline
- AWS CodePipeline Adds Support for Unit and Custom Integration Testing with AWS CodeBuild
- AWS Marketplace: products that can be used for continuous integration
- Continuous Delivery and Continuous Integration
- Software test automation
- Use CodePipeline with AWS CodeBuild to test code and run builds
- Using Canaries (Amazon CloudWatch Synthetics)

REL12-BP04 Test scaling and performance requirements

Use techniques such as load testing to validate that the workload meets scaling and performance requirements.

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

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

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Test scaling and performance requirements. Perform load testing to validate that the workload meets scaling and performance requirements.
  - Distributed Load Testing on AWS: simulate thousands of connected users
  - Apache JMeter
    - Deploy your application in an environment identical to your production environment and execute a load test.
      - Use infrastructure as code concepts to create an environment as similar to your production environment as possible.

Resources

Related documents:

- Distributed Load Testing on AWS: simulate thousands of connected users
- Apache JMeter

REL12-BP05 Test resiliency using chaos engineering

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

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

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

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

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

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

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

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

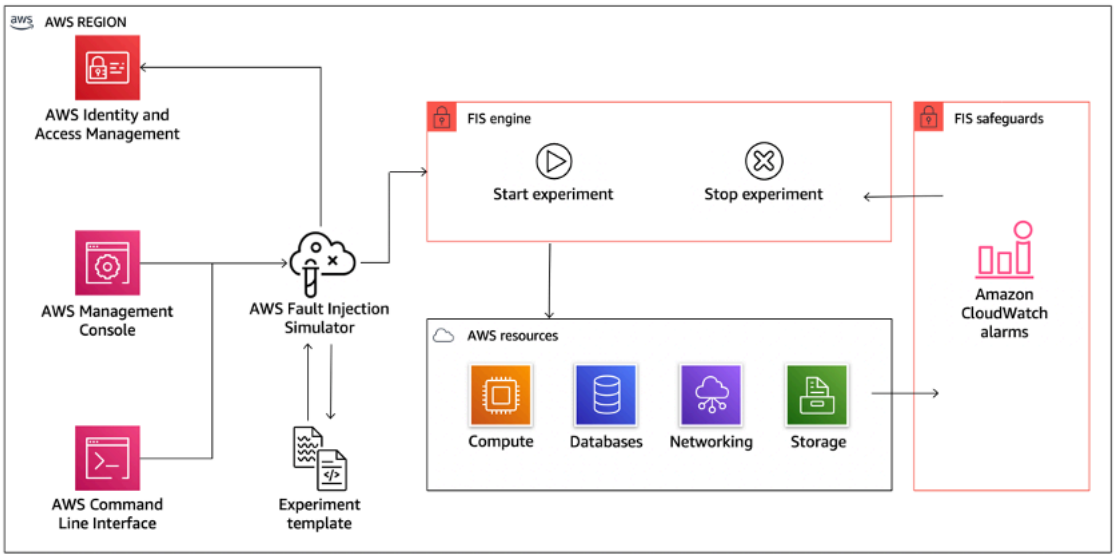


Figure 15: AWS Fault Injection Simulator (AWS FIS) integrates with AWS resources to enable you to run fault injection experiments for your workloads.

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

Common anti-patterns:

- Designing for resiliency, but not testing your recovery.
- Implementing unique recovery paths for each anticipate failure mode.
- Not testing under expected load.
- Not maintaining the resiliency tests as part of the development cycle.

Benefits of establishing this best practice: Injecting failures to test resiliency validates that the recovery procedures will work.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Test resiliency using chaos engineering. Run tests that inject failures regularly into pre-production and production environments. Hypothesize how your workload will react to the failure, then compare your hypothesis to the testing results and iterate if they do not match. Ensure that production testing does not impact users.
  - What is AWS Fault Injection Simulator (AWS FIS)?
  - Principles of Chaos Engineering
  - Well-Architected lab: Level 300: Testing for Resiliency of EC2 RDS and S3
  - Injecting Chaos to Amazon EC2 using AWS Systems Manager
  - AWS re:Invent 2019: Improving resiliency with chaos engineering (DOP309-R1)
    - To inject fault into your workload use Fault Injection Simulator (AWS FIS).
      - What is AWS Fault Injection Simulator (AWS FIS)?
    - Or use open source software.
      - The Chaos ToolKit
      - Shopify Toxiproxy
      - Netflix Chaos Monkey
    - Or use commercial software available through AWS Marketplace.
      - Gremlin
    - Or create your own failure injection code
    - Well-Architected lab: Level 300: Testing for Resiliency of EC2 RDS and S3
    - Injecting Chaos to Amazon EC2 using AWS Systems Manager
    - Test the failure of all components and external dependencies.
      - Simulate conditions that can produce brownouts using extensions to common proxies to introduce latency and dropped messages. You can also create your own implementations to create brownout conditions.

Resources

Related documents:

- Gremlin
- Injecting Chaos to Amazon EC2 using AWS Systems Manager
- Principles of Chaos Engineering
- Resilience Engineering: Learning to Embrace Failure
- The Chaos ToolKit
- What is AWS Fault Injection Simulator (AWS FIS)?

Related videos:

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

Related examples:

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

REL12-BP06 Conduct game days regularly

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

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

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

Common anti-patterns:

- Documenting your procedures, but never exercising them.
- Not including business decision makers in the test exercises.

Benefits of establishing this best practice: Conducting game days regularly ensures that all staff follows the policies and procedures when an actual incident occurs, and validates that those policies and procedures are appropriate.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Schedule game days to regularly exercise your runbooks and playbooks. Game days should involve everyone who would be involved in a production event: business owner, development staff, operational staff, and incident response teams.
  - Run your load or performance tests and then run your failure injection.
  - Look for anomalies in your runbooks and opportunities to exercise your playbooks.
    - If you deviate from your runbooks, refine the runbook or correct the behavior. If you exercise your playbook, identify the runbook that should have been used, or create a new one.

Resources

Related documents:

- What is AWS GameDay?

Related videos:

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

Related examples:

- AWS Well-Architected Labs - Testing Resiliency

Plan for Disaster Recovery (DR)

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

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

Best practices

- REL13-BP01 Define recovery objectives for downtime and data loss (p. 100)
- REL13-BP02 Use defined recovery strategies to meet the recovery objectives (p. 105)
- REL13-BP03 Test disaster recovery implementation to validate the implementation (p. 114)
- REL13-BP04 Manage configuration drift at the DR site or Region (p. 115)
- REL13-BP05 Automate recovery (p. 116)

REL13-BP01 Define recovery objectives for downtime and data loss

The workload has a recovery time objective (RTO) and recovery point objective (RPO).

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

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

RTO and RPO values are important considerations when selecting an appropriate Disaster Recovery (DR) strategy for your workload. These objectives are determined by the business, and then used by technical teams to select and implement a DR strategy.

Desired Outcome:

Every workload has an assigned RTO and RPO, defined based on business impact. The workload is assigned to a predefined tier, defining service availability and acceptable loss of data, with an associated RTO and RPO. If such tiering is not possible then this can be assigned bespoke per workload, with the intent to create tiers later. RTO and RPO are used as one of the primary considerations for selection of a disaster recovery strategy implementation for the workload. Additional considerations in picking a DR strategy are cost constraints, workload dependencies, and operational requirements.

For RTO, understand impact based on duration of an outage. Is it linear, or are there nonlinear implications? (for example. after four hours, you shut down a manufacturing line until the start of the next shift).

A disaster recovery matrix, like the following, can help you understand how workload criticality relates to recovery objectives. (Note that the actual values for the X and Y axes should be customized to your organization needs).

Disaster Recovery Matrix						
		Recovery Point Objective				
		< 1 Minute	< 1 Hour	< 6 Hours	< 1 Day	+ 1 Day
Recovery Time Objective	< 10 Minutes	Critical	Critical	High	Medium	Medium
	< 2 Hours	Critical	High	Medium	Medium	Low
	< 8 Hours	High	Medium	Medium	Low	Low
	< 24 Hours	Medium	Medium	Low	Low	Low
	24 + Hours	Medium	Low	Low	Low	Low

Figure 16: Disaster recovery matrix

Common anti-patterns:

- No defined recovery objectives.
- Selecting arbitrary recovery objectives.
- Selecting recovery objectives that are too lenient and do not meet business objectives.
- Not understanding of the impact of downtime and data loss.
- Selecting unrealistic recovery objectives, such as zero time to recover and zero data loss, which may not be achievable for your workload configuration.

- Selecting recovery objectives more stringent than actual business objectives. This forces DR implementations that are costlier and more complicated than what the workload needs.
- Selecting recovery objectives incompatible with those of a dependent workload.
- Your recovery objectives do not consider regulatory compliance requirements.
- RTO and RPO defined for a workload, but never tested.

Benefits of establishing this best practice: Your recovery objectives for time and data loss are necessary to guide your DR implementation.

Level of risk exposed if this best practice is not established: High

Implementation guidance

For the given workload, you must understand the impact of downtime and lost data on your business. The impact generally grows larger with greater downtime or data loss, but the shape of this growth can differ based on the workload type. For example, you may be able to tolerate downtime for up to an hour with little impact, but after that impact quickly rises. Impact to business manifests in many forms including monetary cost (such as lost revenue), customer trust (and impact to reputation), operational issues (such as missing payroll or decreased productivity), and regulatory risk. Use the following steps to understand these impacts, and set RTO and RPO for your workload.

Implementation Steps

1. Determine your business stakeholders for this workload, and engage with them to implement these steps. Recovery objectives for a workload are a business decision. Technical teams then work with business stakeholders to use these objectives to select a DR strategy.

   Note
   For steps 2 and 3, you can use the the section called “Implementation worksheet” (p. 103).
2. Gather the necessary information to make a decision by answering the questions below.
3. Do you have categories or tiers of criticality for workload impact in your organization?
   a. If yes, assign this workload to a category
   b. If no, then establish these categories. Create five or fewer categories and refine the range of your recovery time objective for each one. Example categories include: critical, high, medium, low. To understand how workloads map to categories, consider whether the workload is mission critical, business important, or non-business driving.
   c. Set workload RTO and RPO based on category. Always choose a category more strict (lower RTO and RPO) than the raw values calculated entering this step. If this results in an unsuitably large change in value, then consider creating a new category.
4. Based on these answers, assign RTO and RPO values to the workload. This can be done directly, or by assigning the workload to a predefined tier of service.
5. Document the disaster recovery plan (DRP) for this workload, which is a part of your organization’s business continuity plan (BCP), in a location accessible to the workload team and stakeholders
   a. Record the RTO and RPO, and the information used to determine these values. Include the strategy used for evaluating workload impact to the business
   b. Record other metrics besides RTO and RPO are you tracking or plan to track for disaster recovery objectives
   c. You will add details of your DR strategy and runbook to this plan when you create these.
6. By looking up the workload criticality in a matrix such as that in Figure 15, you can begin to establish predefined tiers of service defined for your organization.
7. After you have implemented a DR strategy (or a proof of concept for a DR strategy) as per the section called “REL13-BP02 Use defined recovery strategies to meet the recovery objectives” (p. 105), test

this strategy to determine workload actual RTC (Recovery Time Capability) and RPC (Recovery Point Capability). If these do not meet the target recovery objectives, then either work with your business stakeholders to adjust those objectives, or make changes to the DR strategy is possible to meet target objectives.

Primary questions

1. What is the maximum time the workload can be down before severe impact to the business is incurred
   a. Determine the monetary cost (direct financial impact) to the business per minute if workload is disrupted.
   b. Consider that impact is not always linear. Impact can be limited at first, and then increase rapidly past a critical point in time.
2. What is the maximum amount of data that can be lost before severe impact to the business is incurred
   a. Consider this value for your most critical data store. Identify the respective criticality for other data stores.
   b. Can workload data be recreated if lost? If this is operationally easier than backup and restore, then choose RPO based on the criticality of the source data used to recreate the workload data.
3. What are the recovery objectives and availability expectations of workloads that this one depends on (downstream), or workloads that depend on this one (upstream)?
   a. Choose recovery objectives that enable this workload to meet the requirements of upstream dependencies
   b. Choose recovery objectives that are achievable given the recovery capabilities of downstream dependencies. Non-critical downstream dependencies (ones you can “work around”) can be excluded. Or, work with critical downstream dependencies to improve their recovery capabilities where necessary.

Additional questions

Consider these questions, and how they may apply to this workload:

4. Do you have different RTO and RPO depending on the type of outage (Region vs. AZ, etc.)?
5. Is there a specific time (seasonality, sales events, product launches) when your RTO/RPO may change? If so, what is the different measurement and time boundary?
6. How many customers will be impacted if workload is disrupted?
7. What is the impact to reputation if workload is disrupted?
8. What other operational impacts may occur if workload is disrupted? For example, impact to employee productivity if email systems are unavailable, or if Payroll systems are unable to submit transactions.
9. How does workload RTO and RPO align with Line of Business and Organizational DR Strategy?
10. Are there internal contractual obligations for providing a service? Are there penalties for not meeting them?
11. What are the regulatory or compliance constraints with the data?

Implementation worksheet

You can use this worksheet for implementation steps 2 and 3. You may adjust this worksheet to suit your specific needs, such as adding additional questions.

Step 2: Primary questions	Applies to workload?	workload RTO	workload RPO	RTO adjust.	RPO adjust.	Instructions
[1] maximum time the workload can be down						measured in time from start of outage to recovery
[2] maximum amount of data that can be lost						measured in time since last known good restorable dataset
[3a] upstream dependencies						enter the most strict upstream recovery objectives
[3b] downstream dependencies						enter the least strict downstream recovery objectives
[3a] reconciled upstream dependencies						If upstream value is less then current values and downstream value greater,
[3b] reconciled downstream dependencies						then work with dependencies to reconcile and enter reconciled values here
[3] dependencies						lower values to meet upstream dependencies or raise them based on downstream dependency capabilities
Step 2: Additional questions						Indicate if question applies. If it does not apply then skip it
Base RTO/RPO						Carry RTO and RPO values from above down to here
[4] type of outage	[ ]Y / [ ]N					Enter recovery objectives for event type with strictest requirements
[5] specific time-based objectives	[ ]Y / [ ]N					Enter recovery objectives for times with the strictest requirements
[6] customers disrupted	[ ]Y / [ ]N					Graph customers impacted as a function of time down or data lost. Use that to enter the maximum RTO and RPO permissible based on customer impact
[7] reputation impact	[ ]Y / [ ]N					Work with the business to determine maximum RTO and RPO based on impact to reputation
[8] operational impact	[ ]Y / [ ]N					Enter maximum RTO and RPO based on operational impact
[9] organizational alignment	[ ]Y / [ ]N					Enter maximum RTO and RPO for workloads of this type as per LOB and organizational requirements
[10] contractual obligations	[ ]Y / [ ]N					Enter maximum RTO and RPO based on contractual obligations
[11] regulatory compliance	[ ]Y / [ ]N					Enter maximum RTO and RPO based on applicable regulatory compliance
target based on additional questions						Take the minimum value (stricter value) from Q's 4-11 and enter it here
adjusted target						If the objectives on the above line cannot be accommodated, work with stakeholders to loosen constraints, and enter new minimum here
Adjusted RTO/RPO						Enter base RPO/RTO values, or adjusted target, whichever is lower
Step 3						
Map to predefined category or tier						Adjust both values to downward (more strict) to align to nearest defined tier

Worksheet

Level of effort for the Implementation Plan: Low

Resources

Related Best Practices:

- the section called “REL09-BP04 Perform periodic recovery of the data to verify backup integrity and processes” (p. 70)
- the section called “REL13-BP02 Use defined recovery strategies to meet the recovery objectives” (p. 105)
- the section called “REL13-BP03 Test disaster recovery implementation to validate the implementation” (p. 114)

Related documents:

- AWS Architecture Blog: Disaster Recovery Series
- Disaster Recovery of Workloads on AWS: Recovery in the Cloud (AWS Whitepaper)
- Managing resiliency policies with AWS Resilience Hub
- APN Partner: partners that can help with disaster recovery
- AWS Marketplace: products that can be used for disaster recovery

Related videos:

- AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)
- Disaster Recovery of Workloads on AWS

REL13-BP02 Use defined recovery strategies to meet the recovery objectives

Define a disaster recovery (DR) strategy that meets your workload's recovery objectives. Choose a strategy such as: backup and restore; standby (active/passive); or active/active.

A DR strategy relies on the ability to stand up your workload in a recovery site if your primary location becomes unable to run the workload. The most common recovery objectives are RTO and RPO, as discussed in REL13-BP01 Define recovery objectives for downtime and data loss (p. 100).

A DR strategy across multiple Availability Zones (AZs) within a single AWS Region, can provide mitigation against disaster events like fires, floods, and major power outages. If it is a requirement to implement protection against an unlikely event that prevents your workload from being able to run in a given AWS Region, you can use a DR strategy that uses multiple Regions.

When architecting a DR strategy across multiple Regions, you should choose one of the following strategies. They are listed in increasing order of cost and complexity, and decreasing order of RTO and RPO. *Recovery Region* refers to an AWS Region other than the primary one used for your workload.

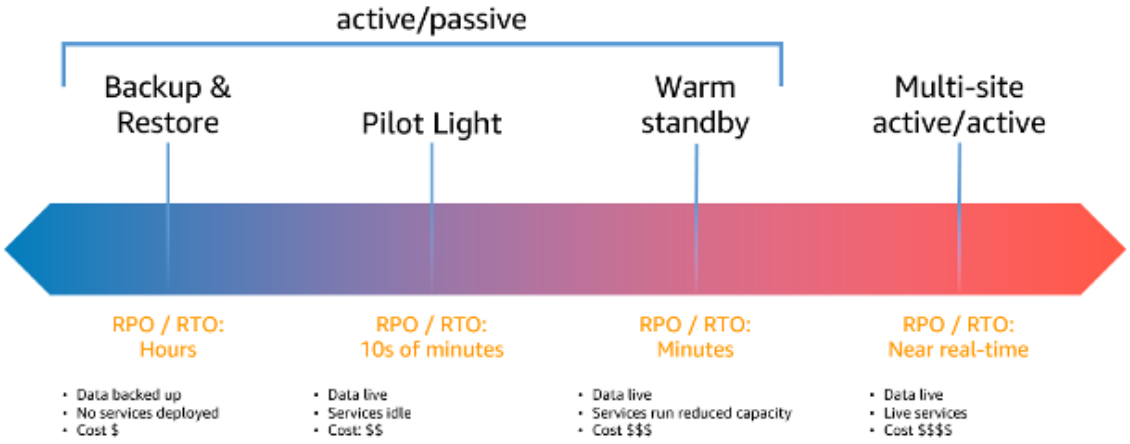


Figure 17: Disaster recovery (DR) strategies

- **Backup and restore** (RPO in hours, RTO in 24 hours or less): Back up your data and applications into the recovery Region. Using automated or continuous backups will enable point in time recovery, which can lower RPO to as low as 5 minutes in some cases. In the event of a disaster, you will deploy your infrastructure (using infrastructure as code to reduce RTO), deploy your code, and restore the backed-up data to recover from a disaster in the recovery Region.
- **Pilot light** (RPO in minutes, RTO in tens of minutes): Provision a copy of your core workload infrastructure in the recovery Region. Replicate your data into the recovery Region and create backups of it there. Resources required to support data replication and backup, such as databases and object storage, are always on. Other elements such as application servers or serverless compute are not deployed, but can be created when needed with the necessary configuration and application code.
- **Warm standby** (RPO in seconds, RTO in minutes): Maintain a scaled-down but fully functional version of your workload always running in the recovery Region. Business-critical systems are fully duplicated and are always on, but with a scaled down fleet. Data is replicated and live in the recovery Region. When the time comes for recovery, the system is scaled up quickly to handle the production load. The more scaled-up the Warm Standby is, the lower RTO and control plane reliance will be. When fully scales this is known as **Hot Standby**.

- **Multi-Region (multi-site) active-active** (RPO near zero, RTO potentially zero): Your workload is deployed to, and actively serving traffic from, multiple AWS Regions. This strategy requires you to synchronize data across Regions. Possible conflicts caused by writes to the same record in two different regional replicas must be avoided or handled, which can be complex. Data replication is useful for data synchronization and will protect you against some types of disaster, but it will not protect you against data corruption or destruction unless your solution also includes options for point-in-time recovery.

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

Desired outcome:

For each workload, there is a defined and implemented DR strategy that enables that workload to achieve DR objectives. DR strategies between workloads make use of reusable patterns (such as the strategies previously described),

Common anti-patterns:

- Implementing inconsistent recovery procedures for workloads with similar DR objectives.
- Leaving the DR strategy to be implemented ad-hoc when a disaster occurs.
- Having no plan for DR.
- Dependency on control plane operations during recovery.

Benefits of establishing this best practice:

- Using defined recovery strategies allows you to use common tooling and test procedures.
- Using defined recovery strategies enables more efficient sharing of knowledge between teams and easier implementation of DR on the workloads they own.

Level of risk exposed if this best practice is not established: High

- Without a planned, implemented, and tested DR strategy, you are unlikely to achieve recovery objectives in the event of a disaster.

Implementation guidance

For each of these steps, see the details below.

1. Determine a DR strategy that will satisfy recovery requirements for this workload.
2. Review the patterns for how the selected DR strategy can be implemented.
3. Assess the resources of your workload, and what their configuration will be in the recovery Region prior to failover (during normal operation).
4. Determine and implement how you will make your recovery Region ready for failover when needed (during a disaster event).
5. Determine and implement how you will reroute traffic to failover when needed (during a disaster event).
6. Design a plan for how your workload will fail back.

Implementation Steps

1. **Determine a DR strategy that will satisfy recovery requirements for this workload.**

Choosing a DR strategy is a trade-off between reducing downtime and data loss (RTO and RPO) versus cost and complexity of implementing the strategy. You should avoid implementing a strategy that is more stringent than it needs to be, as this incurs unnecessary costs.

For example, in the following diagram, the business has determined their maximum permissible RTO as well as the limit of what they can spend on their service restoration strategy. Given the business' objectives, the DR strategies Pilot Light or Warm Standby will satisfy both the RTO and the cost criteria.

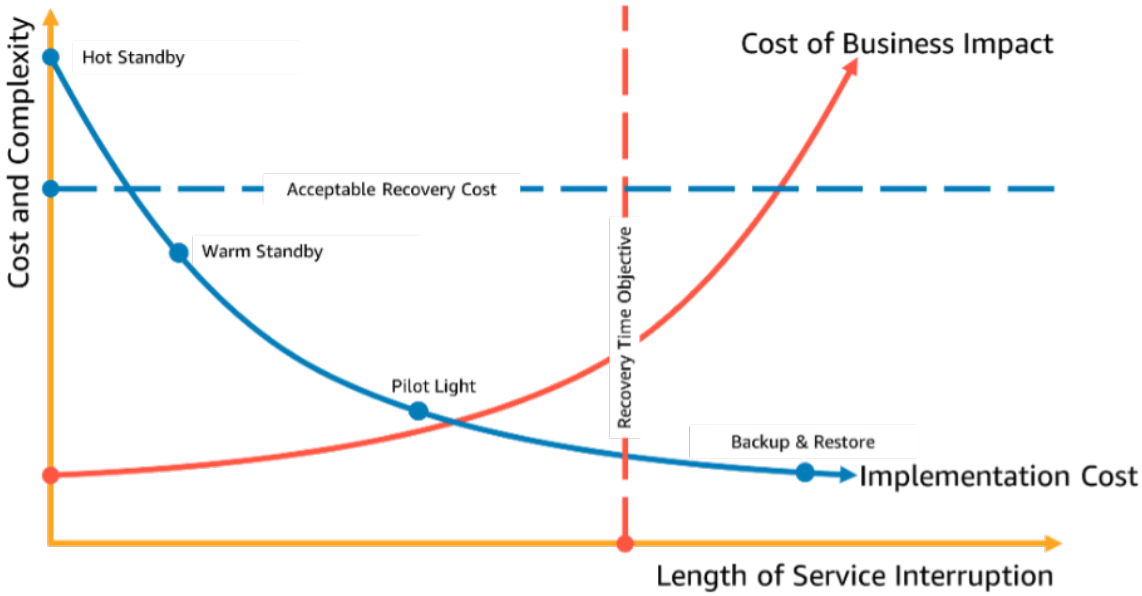


Figure 18: Choosing a DR strategy based on RTO and cost

To learn more see Business Continuity Plan (BCP).

2. **Review the patterns for how the selected DR strategy can be implemented.**

This step is to understand how you will implement the selected strategy. The strategies are explained using AWS Regions as the primary and recovery sites. However, you can also choose to use Availability Zones within a single Region as your DR strategy, which makes use of elements of multiple of these strategies.

In the subsequent steps after this one, you will apply the strategy to your specific workload.

Backup and restore

Backup and restore is the least complex strategy to implement, but will require more time and effort to restore the workload, leading to higher RTO and RPO. It is a good practice to always make backups of your data, and copy these to another site (such as another AWS Region).

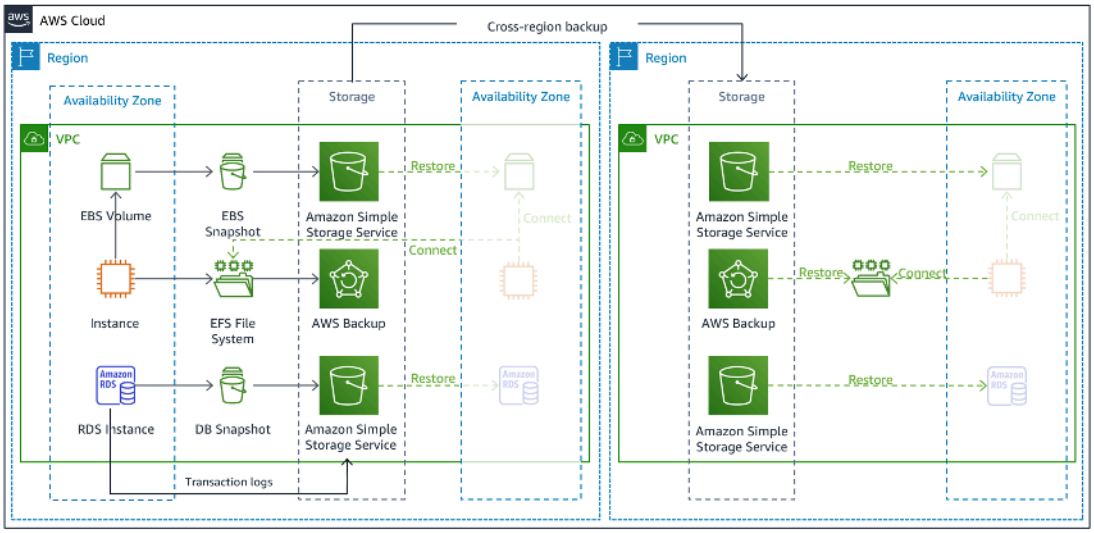


Figure 19: Backup and restore architecture

For more details on this strategy see Disaster Recovery (DR) Architecture on AWS, Part II: Backup and Restore with Rapid Recovery.

Pilot light

With the *pilot light* approach, you replicate your data from your primary Region to your recovery Region. Core resources used for the workload infrastructure are deployed in the recovery Region, however additional resources and any dependencies are still needed to make this a functional stack. For example, in Figure 19, no compute instances are deployed.

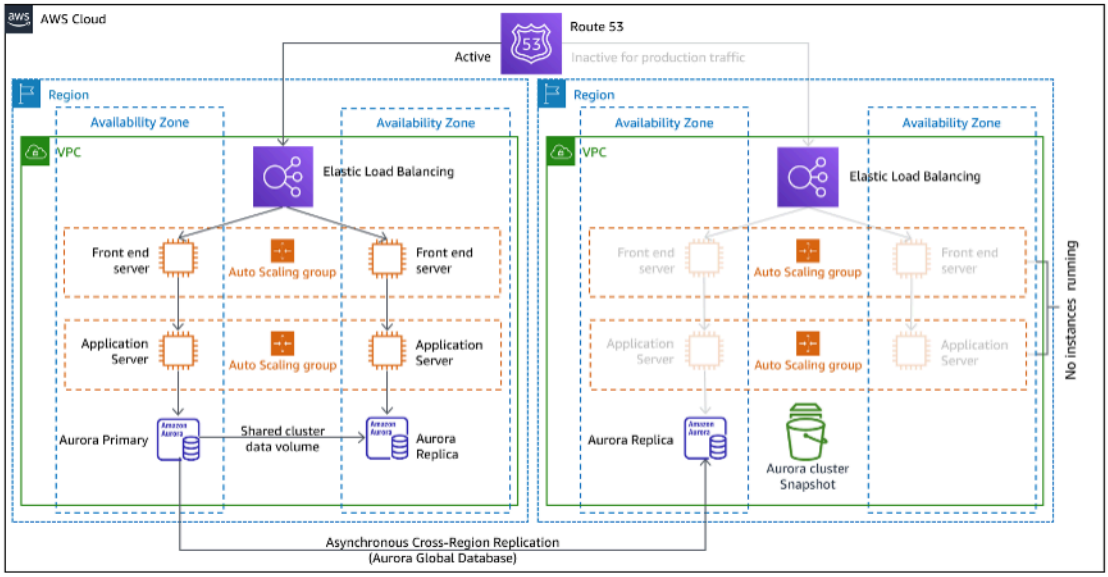


Figure 20: Pilot light architecture

For more details on this strategy see Disaster Recovery (DR) Architecture on AWS, Part III: Pilot Light and Warm Standby.

Warm standby

The *warm standby* approach involves ensuring that there is a scaled down, but fully functional, copy of your production environment in another Region. This approach extends the pilot light concept and decreases the time to recovery because your workload is always-on in another Region. If the recovery Region is deployed at full capacity, then this is known as *hot standby*.

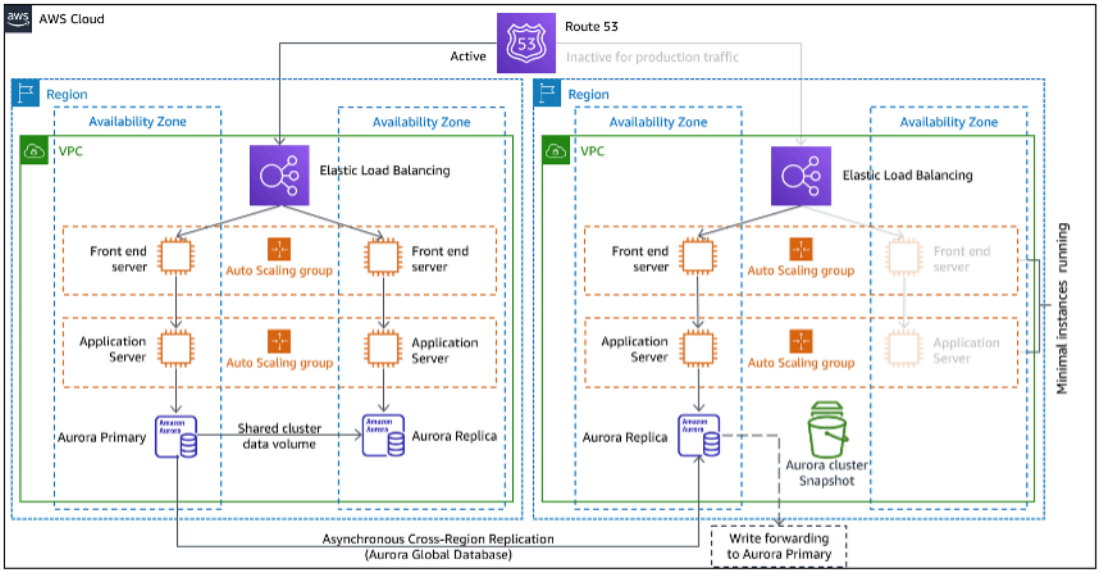


Figure 21: Warm standby architecture

Using warm standby or pilot light requires scaling up resources in the recovery Region. To ensure capacity is available when needed, consider the use for capacity reservations for EC2 instances. If using AWS Lambda, then provisioned concurrency can ensure execution environments so that they are prepared to respond immediately to your function's invocations.

For more details on this strategy, see Disaster Recovery (DR) Architecture on AWS, Part III: Pilot Light and Warm Standby.

Multi-site active/active

You can run your workload simultaneously in multiple Regions as part of a *multi-site active/active* strategy. Multi-site active/active serves traffic from all regions to which it is deployed. Customers may select this strategy for reasons other than DR. It can be used to increase availability, or when deploying a workload to a global audience (to put the endpoint closer to users and/or to deploy stacks localized to the audience in that region). As a DR strategy, if the workload cannot be supported in one of the AWS Regions to which it is deployed, then that Region is evacuated, and the remaining Region(s) are used to maintain availability. Multi-site active/active is the most operationally complex of the DR strategies, and should only be selected when business requirements necessitate it.

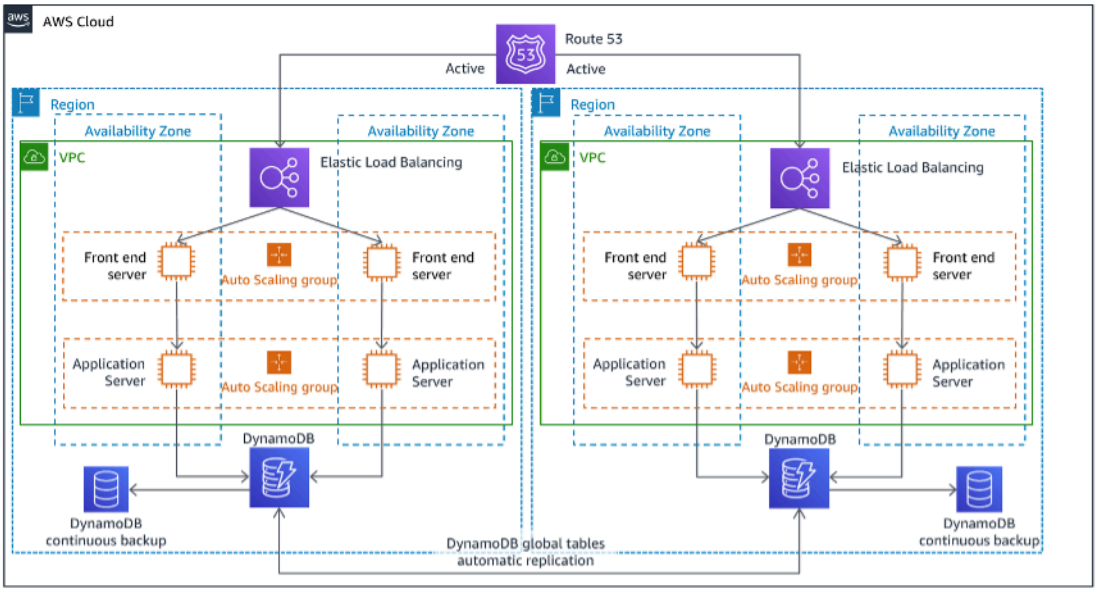


Figure 22: Multi-site active/active architecture

For more details on this strategy see Disaster Recovery (DR) Architecture on AWS, Part IV: Multi-site Active/Active.

Additional practices for protecting data

With all strategies, you must also mitigate against a data disaster. Continuous data replication protects you against some types of disaster, but it may not protect you against data corruption or destruction unless your strategy also includes versioning of stored data or options for point-in-time recovery. You must also back up the replicated data in the recovery site to create point-in-time backups in addition to the replicas.

Using multiple Availability Zones (AZs) within a single AWS Region

When using multiple AZs within a single Region, your DR implementation uses multiple elements of the above strategies. First you must create a high-availability (HA) architecture, using multiple AZs as shown in Figure 22. This architecture makes use of a multi-site active/active approach, as the Amazon EC2 instances and the Elastic Load Balancer have resources deployed in multiple AZs, actively handing requests. The architecture also demonstrates hot standby, where if the primary Amazon RDS instance fails (or the AZ itself fails), then the standby instance is promoted to primary.

Figure 23: Multi-AZ architecture

In addition to this HA architecture, you need to add backups of all data required to run your workload. This is especially important for data that is constrained to a single zone such as Amazon EBS volumes or Amazon Redshift clusters. If an AZ fails, you will need to restore this data to another AZ. Where possible, you should also copy data backups to another AWS Region as an additional layer of protection.

An less common alternative approach to single Region, multi-AZ DR is illustrated in the blog post, Building highly resilient applications using Amazon Route 53 Application Recovery Controller, Part 1: Single-Region stack. Here, the strategy is to maintain as much isolation between the AZs as possible, like how Regions operate. Using this alternative strategy, you can choose an active/active or active/passive approach.

Note: Some workloads have regulatory data residency requirements. If this applies to your workload in a locality that currently has only one AWS Region, then multi-Region will not suit your business needs. Multi-AZ strategies provide good protection against most disasters.

3. **Assess the resources of your workload, and what their configuration will be in the recovery Region prior to failover (during normal operation).**

For infrastructure and AWS resources us infrastructure as code such as AWS CloudFormation or third-party tools like Hashicorp Terraform. To deploy across multiple accounts and Regions with a single operation you can use AWS CloudFormation StackSets. For Multi-site active/active and Hot Standby strategies, the deployed infrastructure in your recovery Region has the same resources as your primary Region. For Pilot Light and Warm Standby strategies, the deployed infrastructure will require additional actions to become production ready. Using CloudFormation parameters and conditional logic, you can control whether a deployed stack is active or standby with a single template. An example of such a CloudFormation template is included in this blog post.

All DR strategies require that data sources are backed up within the AWS Region, and then those backups are copied to the recovery Region. AWS Backup provides a centralized view where you can configure, schedule, and monitor backups for these resources. For Pilot Light, Warm Standby, and Multi-site active/active, you should also replicate data from the primary Region to data resources in the recovery Region, such as Amazon Relational Database Service (Amazon RDS) DB instances or Amazon DynamoDB tables. These data resources are therefore live and ready to serve requests in the recovery Region.

To learn more about how AWS services operate across Regions, see this blog series on Creating a Multi-Region Application with AWS Services.

4. **Determine and implement how you will make your recovery Region ready for failover when needed (during a disaster event).**

For Multi-site active/active, failover means evacuating a Region, and relying on the remaining active Regions. In general, those Regions are ready to accept traffic. For Pilot Light and Warm Standby strategies, your recovery actions will need to deploy the missing resources, such as the EC2 instances in Figure 19, plus any other missing resources.

For all of the above strategies you may need to promote read-only instances of databases to become the primary read/write instance.

For backup and restore, restoring data from backup creates resources for that data such as EBS volumes, RDS DB instances, and DynamoDB tables. You also need to restore the infrastructure and deploy code. You can use AWS Backup to restore data in the recovery Region. See REL09-BP01 Identify and back up all data that needs to be backed up, or reproduce the data from sources (p. 64) for more details. Rebuilding the infrastructure includes creating resources like EC2 instances in addition to the Amazon Virtual Private Cloud (Amazon VPC), subnets, and security groups needed. You can automate much of the restoration process. To learn how, see this blog post.

5. **Determine and implement how you will reroute traffic to failover when needed (during a disaster event).**

This failover operation can be initiated either automatically or manually. Automatically initiated failover based on health checks or alarms should be used with caution since an unnecessary failover (false alarm) incurs costs such as non-availability and data loss. Manually initiated failover is therefore often used. In this case, you should still automate the steps for failover, so that the manual initiation is like the push of a button.

There are several traffic management options to consider when using AWS services. One option is to use Amazon Route 53. Using Amazon Route 53, you can associate multiple IP endpoints in one or more AWS Regions with a Route 53 domain name. To implement manually initiated failover you can use Amazon Route 53 Application Recovery Controller, which provides a highly available data plane API to reroute traffic to the recovery Region. When implementing failover, use data plane operations and avoid control plane ones as described in REL11-BP04 Rely on the data plane and not the control plane during recovery (p. 89).

To learn more about this and other options see this section of the Disaster Recovery Whitepaper.

6. **Design a plan for how your workload will fail back.**

Failback is when you return workload operation to the primary Region, after a disaster event has abated. Provisioning infrastructure and code to the primary Region generally follows the same steps as were initially used, relying on infrastructure as code and code deployment pipelines. The challenge with failback is restoring data stores, and ensuring their consistency with the recovery Region in operation.

In the failed over state, the databases in the recovery Region are live and have the up-to-date data. The goal then is to re-synchronize from the recovery Region to the primary Region, ensuring it is up-to-date.

Some AWS services will do this automatically. If using Amazon DynamoDB global tables, even if the table in the primary Region had become not available, when it comes back online, DynamoDB resumes propagating any pending writes. If using Amazon Aurora Global Database and using managed planned failover, then Aurora global database's existing replication topology is maintained. Therefore, the former read/write instance in the primary Region will become a replica and receive updates from the recovery Region.

In cases where this is not automatic, you will need to re-establish the database in the primary Region as a replica of the database in the recovery Region. In many cases this will involve deleting the old primary database, and creating new replicas. For example, for instructions on how to do this with Amazon Aurora Global Database assuming an *unplanned* failover see this lab: Fail Back a Global Database.

After a failover, if you can continue running in your recovery Region, consider making this the new primary Region. You would still do all the above steps to make the former primary Region into a recovery Region. Some organizations do a scheduled rotation, swapping their primary and recovery Regions periodically (for example every three months).

All of the steps required to fail over and fail back should be maintained in a playbook that is available to all members of the team, and is periodically reviewed.

Level of effort for the Implementation Plan: High

Resources

Related Best Practices:

- the section called “REL09-BP01 Identify and back up all data that needs to be backed up, or reproduce the data from sources” (p. 64)
- the section called “REL11-BP04 Rely on the data plane and not the control plane during recovery” (p. 89)
- the section called “REL13-BP01 Define recovery objectives for downtime and data loss” (p. 100)

Related documents:

- AWS Architecture Blog: Disaster Recovery Series
- Disaster Recovery of Workloads on AWS: Recovery in the Cloud (AWS Whitepaper)
- Disaster recovery options in the cloud
- Build a serverless multi-region, active-active backend solution in an hour
- Multi-region serverless backend — reloaded
- RDS: Replicating a Read Replica Across Regions
- Route 53: Configuring DNS Failover
- S3: Cross-Region Replication
- What Is AWS Backup?
- What is Route 53 Application Recovery Controller?
- AWS Elastic Disaster Recovery
- HashiCorp Terraform: Get Started - AWS
- APN Partner: partners that can help with disaster recovery
- AWS Marketplace: products that can be used for disaster recovery

Related videos:

- Disaster Recovery of Workloads on AWS
- AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)
- Get Started with AWS Elastic Disaster Recovery | Amazon Web Services

Related examples:

- AWS Well-Architected Labs - Disaster Recovery - Series of workshops illustrating the DR strategies

REL13-BP03 Test disaster recovery implementation to validate the implementation

Regularly test failover to your recovery site to ensure proper operation, and that RTO and RPO are met.

A pattern to avoid is developing recovery paths that are rarely exercised. For example, you might have a secondary data store that is used for read-only queries. When you write to a data store and the primary fails, you might want to fail over to the secondary data store. If you don't frequently test this failover, you might find that your assumptions about the capabilities of the secondary data store are incorrect. The capacity of the secondary, which might have been sufficient when you last tested, might 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 exercise that failure in production to convince yourself that the recovery path works. In the example we just discussed, you should fail over to the standby regularly, regardless of need.

Common anti-patterns:

- Never exercise failovers in production.

Benefits of establishing this best practice: Regularly testing you disaster recovery plan ensures that it will work when it needs to, and that your team knows how to execute the strategy.

Level of risk exposed if this best practice is not established: High

Implementation guidance

- Engineer your workloads for recovery. Regularly test your recovery paths Recovery Oriented Computing 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. Exercise the recovery path to ensure that you can accomplish the recovery in the specified time to the specified state. Use your runbooks during this recovery to document problems and find solutions for them before the next test.
  - The Berkeley/Stanford recovery-oriented computing project
- Use AWS Elastic Disaster Recovery to implement and launch drill instances for your DR strategy.
  - AWS Elastic Disaster Recovery Preparing for Failover
  - What is Elastic Disaster Recovery?
  - AWS Elastic Disaster Recovery

Resources

Related documents:

- APN Partner: partners that can help with disaster recovery
- AWS Architecture Blog: Disaster Recovery Series
- AWS Marketplace: products that can be used for disaster recovery
- Elastic Disaster Recovery
- Disaster Recovery of Workloads on AWS: Recovery in the Cloud (AWS Whitepaper)
- AWS Elastic Disaster Recovery Preparing for Failover

- The Berkeley/Stanford recovery-oriented computing project
- What is AWS Fault Injection Simulator?

Related videos:

- AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)
- AWS re:Invent 2019: Backup-and-restore and disaster-recovery solutions with AWS (STG208)

Related examples:

- AWS Well-Architected Labs - Testing for Resiliency

REL13-BP04 Manage configuration drift at the DR site or Region

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

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

Common anti-patterns:

- Failing to make updates in your recovery locations, when you make configuration or infrastructure changes in your primary locations.
- Not considering potential limitations (like service differences) in your primary and recovery locations.

Benefits of establishing this best practice: Ensuring that your DR environment is consistent with your existing environment guarantees complete recovery.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Ensure that your delivery pipelines deliver to both your primary and backup sites. Delivery pipelines for deploying applications into production must distribute to all the specified disaster recovery strategy locations, including dev and test environments.
- Enable AWS Config to track potential drift locations. Use AWS Config rules to create systems that enforce your disaster recovery strategies and generate alerts when they detect drift.
  - Remediating Noncompliant AWS Resources by AWS Config Rules
  - AWS Systems Manager Automation
- Use AWS CloudFormation to deploy your infrastructure. AWS CloudFormation can detect drift between what your CloudFormation templates specify and what is actually deployed.
  - AWS CloudFormation: Detect Drift on an Entire CloudFormation Stack

Resources

Related documents:

- APN Partner: partners that can help with disaster recovery

- AWS Architecture Blog: Disaster Recovery Series
- AWS CloudFormation: Detect Drift on an Entire CloudFormation Stack
- AWS Marketplace: products that can be used for disaster recovery
- AWS Systems Manager Automation
- Disaster Recovery of Workloads on AWS: Recovery in the Cloud (AWS Whitepaper)
- How do I implement an Infrastructure Configuration Management solution on AWS?
- Remediating Noncompliant AWS Resources by AWS Config Rules

Related videos:

- AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)

REL13-BP05 Automate recovery

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

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

For workloads on existing physical or virtual data centers or private clouds, AWS Elastic Disaster Recovery allows organizations to set up an automated disaster recovery strategy in AWS. Elastic Disaster Recovery also supports cross-Region and cross-Availability Zone disaster recovery in AWS.

Common anti-patterns:

- Implementing identical automated failover and failback can cause flapping when a failure occurs.

Benefits of establishing this best practice: Automated recovery reduces your recovery time by eliminating the opportunity for manual errors.

Level of risk exposed if this best practice is not established: Medium

Implementation guidance

- Automate recovery paths. For short recovery times, follow your disaster recovery plan to get your IT systems back online quickly in the case of a disruption.
  - Use Elastic Disaster Recovery for automated Failover and Failback. Elastic Disaster Recovery continuously replicates your machines (including operating system, system state configuration, databases, applications, and files) into a low-cost staging area in your target AWS account and preferred Region. In the case of a disaster, after choosing to recover using Elastic Disaster Recovery, Elastic Disaster Recovery automates the conversion of your replicated servers into fully provisioned workloads in your recovery Region on AWS.
    - Using Elastic Disaster Recovery for Failover and Failback
    - AWS Elastic Disaster Recovery resources

Resources

Related documents:

- APN Partner: partners that can help with disaster recovery
- AWS Architecture Blog: Disaster Recovery Series
- AWS Marketplace: products that can be used for disaster recovery
- AWS Systems Manager Automation
- AWS Elastic Disaster Recovery
- Disaster Recovery of Workloads on AWS: Recovery in the Cloud (AWS Whitepaper)

Related videos:

- AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)

Example implementations for availability goals

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

Topic	For more information, see this section
Monitor resources	Monitor workload resources (p. 43)
Adapt to changes in demand	Design your workload to adapt to changes in demand (p. 52)
Implement change	Implement change (p. 57)
Back up data	Back up data (p. 64)
Architect for resiliency	Use fault isolation to protect your workload (p. 72) Design your workload to withstand component failures (p. 83)
Test resiliency	Test reliability (p. 93)
Plan for disaster recovery (DR)	Plan for Disaster Recovery (DR) (p. 100)

Dependency selection

We have chosen to use Amazon EC2 for our applications. We will show how using Amazon RDS and multiple Availability Zones improves the availability of our applications. We will use Amazon Route 53 for DNS. When we use multiple Availability Zones, we will use Elastic Load Balancing. Amazon S3 is used for backups and static content. As we design for higher reliability, we must use services with higher availability themselves. See Appendix A: Designed-For Availability for Select AWS Services (p. 137) for the design goals for the respective AWS services.

Single-Region scenarios

Topics

2 9s (99%) scenario

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

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

Monitor resources

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

Adapt to changes in demand

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

Implement change

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

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

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

Back up data

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

Architect for resiliency

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

Test resiliency

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

Plan for disaster recovery (DR)

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

Availability design goal

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

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

Summary

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

3 9s (99.9%) scenario

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

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

Monitor resources

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

Adapt to changes in demand

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

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

Implement change

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

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

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

Back up data

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

Architect for resiliency

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

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

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

Test resiliency

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

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

Plan for disaster recovery (DR)

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

Availability design goal

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

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

Summary

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

4 9s (99.99%) scenario

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

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

Monitor resources

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

Adapt to changes in demand

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

Implement change

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

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

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

Back up data

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

Architect for resiliency

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

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

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

Test resiliency

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

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

Plan for disaster recovery (DR)

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

Availability design goal

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

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

Summary

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

Multi-Region scenarios

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

Topics

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

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

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

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

Monitor resources

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

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

Adapt to changes in demand

Same as the 4 9s scenario.

Implement change

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

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

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

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

Back up data

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

Architect for resiliency

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

Test resiliency

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

Plan for disaster recovery (DR)

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

Availability design goal

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

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

Summary

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

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

5 9s (99.999%) or higher scenario with a recovery time under one minute

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

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

Monitor resources

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

Adapt to changes in demand

Same as the 3½ 9s scenario.

Implement change

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

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

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

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

Back up data

Same as the 3½ 9s scenario.

Architect for resiliency

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

Test resiliency

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

Plan for disaster recovery (DR)

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

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

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

Availability design goal

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

Summary

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

Resources

Documentation

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

- AWS Architecture Center

Labs

- AWS Well-Architected Reliability Labs

External Links

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

Books

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

Conclusion

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

Contributors

Contributors to this document include:

- Seth Eliot, Principal Reliability Solutions Architect – Well-Architected, Amazon Web Services
- Mahanth Jayadeva, Solutions Architect – Well-Architected, Amazon Web Services
- Amulya Sharma, Principal Solutions Architect, Amazon Web Services
- Jason DiDomenico, Senior Solutions Architect – Cloud Foundations, Amazon Web Services
- Marcin Bednarz, Principal Solutions Architect, Amazon Web Services
- Tyler Applebaum, Senior Solutions Architect, Amazon Web Services
- Rodney Lester, Principal Solutions Architect – App Modernization, Amazon Web Services
- Joe Chapman, Senior Solutions Architect, Amazon Web Services
- Adrian Hornsby, Principal System Development Engineer, Amazon Web Services
- Kevin Miller, Vice President – S3, Amazon Web Services
- Shannon Richards, Principal Technical Program Manager, Amazon Web Services

Further reading

For additional information, see:

- AWS Well-Architected Framework
- AWS Architecture Center

Document revisions

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

Change	Description	Date
Whitepaper updated (p. 134)	Best practices expanded and improvement plans added.	October 20, 2022
Whitepaper updated (p. 134)	Added two new best practices to Reliability Pillar in sections **Use Fault Isolation to Protect Your Workload** and **Design your Workload to Withstand Component Failures**.	May 5, 2022
Minor update (p. 1)	Added Sustainability Pillar to introduction.	December 2, 2021
Whitepaper updated (p. 134)	Update Disaster Recovery guidance to include Route 53 Application Recovery Controller. Add references to DevOps Guru. Update several Resource links, and other minor editorial changes.	October 26, 2021
Minor update (p. 134)	Added information about AWS Fault Injection Simulator (AWS FIS).	March 15, 2021
Minor update (p. 134)	Minor text update.	January 4, 2021
Whitepaper updated (p. 134)	Updated Appendix A to update the Availability Design Goal for Amazon SQS, Amazon SNS, and Amazon MQ; Re-order rows in table for easier lookup; Improve explanation of differences between availability and disaster recovery and how they both contribute to resiliency; Expand coverage of multi-region architectures (for availability) and multi-region strategies (for disaster recovery); Update referenced book to latest version; Expand availability calculations to include request-based calculation, and shortcut calculations; Improve description for Game Days	December 7, 2020
Minor update (p. 134)	Updated Appendix A to update the Availability Design Goal for AWS Lambda	October 27, 2020

Minor update (p. 134)	Updated Appendix A to add the Availability Design Goal for AWS Global Accelerator	July 24, 2020
Updates for new Framework (p. 134)	Substantial updates and new/revised content, including: Added "Workload Architecture" best practices section, re-organized best practices into Change Management and Failure Management sections, updated Resources, updated to include latest AWS resources and services such as AWS Global Accelerator, AWS Service Quotas, and AWS Transit Gateway, added/updated definitions for Reliability, Availability, Resiliency, better aligned whitepaper to the AWS Well-Architected Tool (questions and best practices) used for Well-Architected Reviews, re-order design principles, moving **Automatically recover from failure** before **Test recovery procedures**, updated diagrams and formats for equations, removed Key Services sections and instead integrated references to key AWS services into the best practices.	July 8, 2020
Minor update (p. 134)	Fixed broken link	October 1, 2019
Whitepaper updated (p. 134)	Appendix A updated	April 1, 2019
Whitepaper updated (p. 134)	Added specific AWS Direct Connect networking recommendations and additional service design goals	September 1, 2018
Whitepaper updated (p. 134)	Added Design Principles and Limit Management sections. Updated links, removed ambiguity of upstream/downstream terminology, and added explicit references to the remaining Reliability Pillar topics in the availability scenarios.	June 1, 2018
Whitepaper updated (p. 134)	Changed DynamoDB Cross Region solution to DynamoDB Global Tables. Added service design goals	March 1, 2018

Minor updates (p. 134)	Minor correction to availability calculation to include application availability	December 1, 2017
Whitepaper updated (p. 134)	Updated to provide guidance on high availability designs, including concepts, best practice and example implementations.	November 1, 2017
Initial publication (p. 134)	Reliability Pillar - AWS Well-Architected Framework published.	November 1, 2016

Appendix A: Designed-For Availability for Select AWS Services

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

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

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

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

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

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