AWS Well-Architected Framework

Reliability Pillar

Reliability Pillar: AWS Well-Architected Framework

Table of Contents

Reliability Pillar - AWS Well-Architected Framework

Publication date: **November 6, 2024** (*Document revisions*)

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, cost-effective, and sustainable 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

Shared Responsibility Model for Resiliency

Resiliency is a shared responsibility between AWS and you. It is important that you understand how disaster recovery (DR) and availability, as part of resiliency, operate under this shared model.

AWS responsibility - Resiliency of the cloud

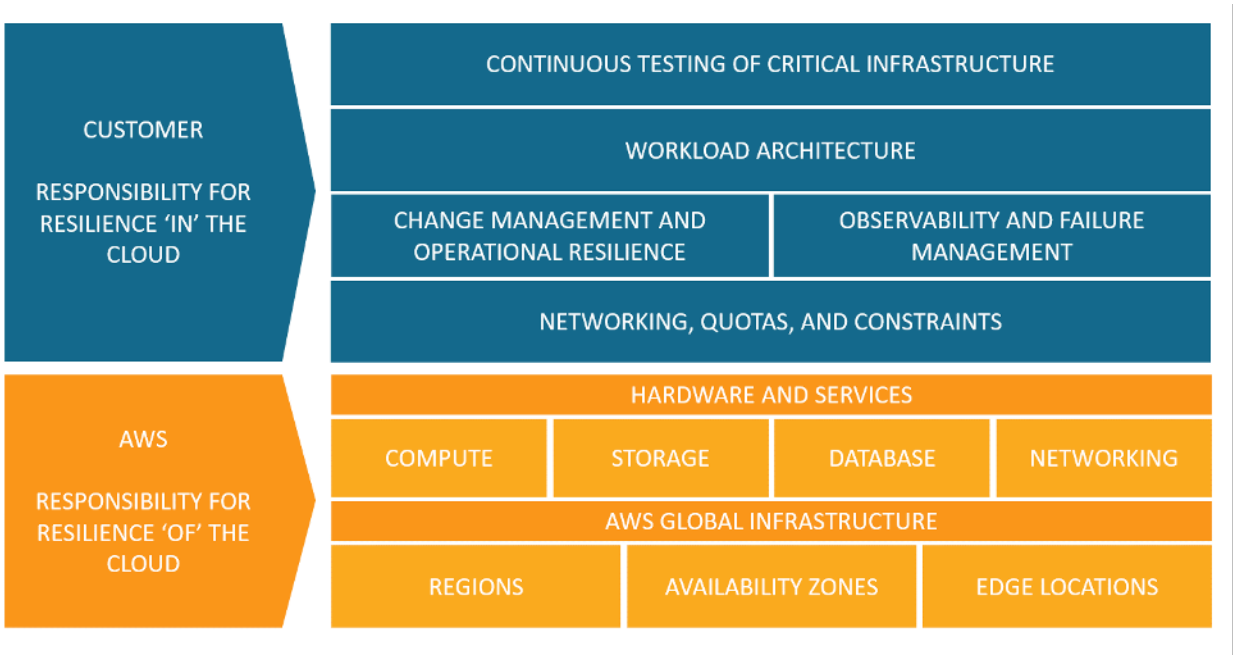
AWS is responsible for resiliency of the infrastructure that runs all of the services offered in the AWS Cloud. This infrastructure comprises the hardware, software, networking, and facilities that run AWS Cloud services. AWS uses commercially reasonable efforts to make these AWS Cloud services available, ensuring service availability meets or exceeds AWS Service Level Agreements (SLAs).

The AWS Global Cloud Infrastructure is designed to allow customers to build highly resilient workload architectures. Each AWS Region is fully isolated and consists of multiple Availability Zones, which are physically isolated partitions of infrastructure. Availability Zones isolate faults that could impact workload resilience, preventing them from impacting other zones in the Region. But at the same time, all zones in an AWS Region are interconnected with high-bandwidth, low-latency networking, over fully redundant, dedicated metro fiber providing high-throughput, low-latency networking between zones. All traffic between zones is encrypted. The network performance is sufficient to accomplish synchronous replication between zones. When an application is partitioned across AZs, companies are better isolated and protected from issues such as power outages, lightning strikes, tornadoes, hurricanes, and more.

Customer responsibility - Resiliency in the cloud

Your responsibility is determined by the AWS Cloud services that you select. This determines the amount of configuration work you must perform as part of your resiliency responsibilities. For example, a service such as Amazon Elastic Compute Cloud (Amazon EC2) requires the customer to perform all of the necessary resiliency configuration and management tasks. Customers that deploy Amazon EC2 instances are responsible for deploying Amazon EC2 instances across multiple locations (such as AWS Availability Zones), implementing self-healing using services like Auto Scaling, and using resilient workload architecture best practices for applications installed on the instances. For managed services, such as Amazon S3 and Amazon DynamoDB, AWS operates the infrastructure layer, the operating system, and platforms, and customers access the endpoints to store and retrieve data. You are responsible for managing resiliency of your data including backup, versioning, and replication strategies.

Deploying your workload across multiple Availability Zones in an AWS Region is part of a high availability strategy designed to protect workloads by isolating issues to one Availability Zone, which uses the redundancy of the other Availability Zones to continue serving requests. A Multi-AZ architecture is also part of a DR strategy designed to make workloads better isolated and protected from issues such as power outages, lightning strikes, tornadoes, earthquakes, and more. DR strategies may also make use of multiple AWS Regions. For example, in an active/passive configuration, service for the workload fails over from its active Region to its DR Region if the active Region can no longer serve requests.



Responsibility for resilience in and of the cloud for customers and AWS.

You can use AWS services to achieve your resilience objectives. As a customer, you are responsible for management of the following aspects of your system to achieve resilience in the cloud. For more detail on each service in particular, see AWS documentation.

Networking, quotas, and constraints

- Best practices for this area of the shared responsibility model are described in detail under Foundations.
- Plan your architecture with adequate room to scale and understand the service quotas and constraints of the services you include, based on expected load request increases where applicable.
- Design your network topology to be highly available, redundant, and scalable.

Change management and operational resilience

- Change management includes how to introduce and manage change in your environment. Implementing change requires building and keeping runbooks up to date and deployment strategies for your application and infrastructure.
- A resilient strategy for monitoring workload resources considers all components, including both technical and business metrics, notifications, automation, and analysis.
- Workloads in the cloud must adapt to changes in demand scaling in reaction to impairments or fluctuations in usage.

Observability and failure management

- Observing failures through monitoring is required to automate healing so that your workloads can withstand component failures.
- Failure management requires backing up data, applying best practices to allow your workload to withstand component failures, and planning for disaster recovery.

Workload architecture

- Your workload architecture includes how you design services around business domains, apply SOA and distributed system design to prevent failures, and build in capabilities like throttling, retries, queue management, timeouts, and emergency levers.

- Rely on proven AWS solutions, the Amazon Builders Library, and serverless patterns to align with best practices and jump start implementations.
- Use continuous improvement to decompose your system into distributed services to scale and innovate faster. Use AWS microservices guidance and managed service options to simplify and accelerate your ability to introduce change and innovate.

Continuous testing of critical infrastructure

- Testing reliability means testing at the functional, performance, and chaos levels, as well as adopting incident analysis and game day practices to build expertise in resolving issues that are not well understood.
- For both cloud all-in and hybrid applications, knowing how your application behaves when issues arise or components go down allows you to quickly and reliably recover from outages.
- Create and document repeatable experiments to understand how your system behaves when things don't work as expected. These tests will prove effectiveness of your overall resilience and provide a feedback loop for your operational procedures before facing real failure scenarios.

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 run 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).
- **Manage change through 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
- Availability
- Disaster Recovery (DR) objectives

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%	3 days 15 hours	Batch processing, data extraction, transfer, and load jobs
99.9%	8 hours 45 minutes	Internal tools like knowledge management, project tracking
99.95%	4 hours 22 minutes	Online commerce, point of sale
99.99%	52 minutes	Video delivery, broadcast **workloads**

Availability	Maximum Unavailability (per year)	Application Categories
99.999%	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:

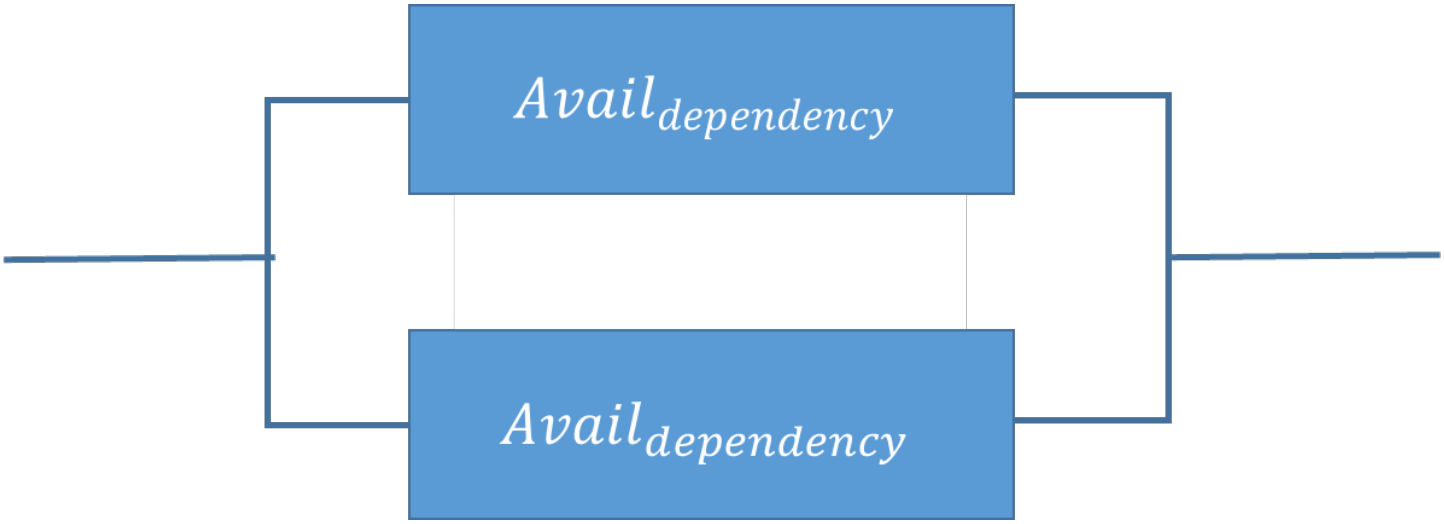
$Avail_{invok} \times Avail_{dep1} \times Avail_{dep2} = Avail_{workload}$

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%:



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

$99.9999\% = 100\% - (0.1\% \times 0.1\%)$

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. 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 to 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 performed 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-BP05 Automate quota management
- REL01-BP06 Ensure that a sufficient gap exists between the current quotas and the maximum usage to accommodate failover

REL01-BP01 Aware of service quotas and constraints

Be aware of your default quotas and manage your quota increase requests for your workload architecture. Know which cloud resource constraints, such as disk or network, are potentially impactful.

Desired outcome: Customers can prevent service degradation or disruption in their AWS accounts by implementing proper guidelines for monitoring key metrics, infrastructure reviews, and automation remediation steps to verify that services quotas and constraints are not reached that could cause service degradation or disruption.

Common anti-patterns:

- Deploying a workload without understanding the hard or soft quotas and their limits for the services used.
- Deploying a replacement workload without analyzing and reconfiguring the necessary quotas or contacting Support in advance.
- Assuming that cloud services have no limits and the services can be used without consideration to rates, limits, counts, quantities.
- Assuming that quotas will automatically be increased.
- Not knowing the process and timeline of quota requests.
- Assuming that the default cloud service quota is the identical for every service compared across regions.
- Assuming that service constraints can be breached and the systems will auto-scale or add increase the limit beyond the resource's constraints
- Not testing the application at peak traffic in order to stress the utilization of its resources.
- Provisioning the resource without analysis of the required resource size.
- Overprovisioning capacity by choosing resource types that go well beyond actual need or expected peaks.
- Not assessing capacity requirements for new levels of traffic in advance of a new customer event or deploying a new technology.

Benefits of establishing this best practice: Monitoring and automated management of service quotas and resource constraints can proactively reduce failures. Changes in traffic patterns for a customer's service can cause a disruption or degradation if best practices are not followed. By monitoring and managing these values across all regions and all accounts, applications can have improved resiliency under adverse or unplanned events.

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

Implementation guidance

Service Quotas is an AWS service that helps you manage your quotas for over 250 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 using 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).

Some service limits, like rate limits on throttled APIs are set within the Amazon API Gateway itself by configuring a usage plan. Some limits that are set as configuration on their respective services include Provisioned IOPS, Amazon RDS storage allocated, and Amazon EBS volume allocations. Amazon Elastic Compute Cloud has its own service limits dashboard that can help you manage your instance, Amazon Elastic Block Store, 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 Support to see if there are mitigations.

Service quotas can be Region specific or can also be global in nature. Using an AWS service that reaches its quota will not act as expected in normal usage and may cause service disruption or degradation. For example, a service quota limits the number of DL Amazon EC2 instances used in a Region. That limit may be reached during a traffic scaling event using Auto Scaling groups (ASG).

Service quotas for each account should be assessed for usage on a regular basis to determine what the appropriate service limits might be for that account. These service quotas exist as operational guardrails, to prevent accidentally provisioning more resources than you need. They also serve to limit request rates on API operations to protect services from abuse.

Service constraints are different from service quotas. Service constraints represent a particular resource's limits as defined by that resource type. These might be storage capacity (for example, gp2 has a size limit of 1 GB - 16 TB) or disk throughput. It is essential that a resource type's constraint be engineered and constantly assessed for usage that might reach its limit. If a

constraint is reached unexpectedly, the account's applications or services may be degraded or disrupted.

If there is a use case where service quotas impact an application's performance and they cannot be adjusted to required needs, contact Support to see if there are mitigations. For more detail on adjusting fixed quotas, see REL01-BP03 Accommodate fixed service quotas and constraints through architecture.

There are a number of AWS services and tools to help monitor and manage Service Quotas. The service and tools should be leveraged to provide automated or manual checks of quota levels.

- AWS Trusted Advisor offers a service quota check that displays your usage and quotas for some aspects of some services. It can aid in identifying services that are near quota.
- AWS Management Console provides methods to display services quota values, manage, request new quotas, monitor status of quota requests, and display history of quotas.
- AWS CLI and CDKs offer programmatic methods to automatically manage and monitor service quota levels and usage.

Implementation steps

For Service Quotas:

- Review AWS Service Quotas.
- To be aware of your existing service quotas, determine the services (like IAM Access Analyzer) that are used. There are approximately 250 AWS services controlled by service quotas. Then, determine the specific service quota name that might be used within each account and Region. There are approximately 3000 service quota names per Region.
- Augment this quota analysis with AWS Config to find all AWS resources used in your AWS accounts.
- Use AWS CloudFormation data to determine your AWS resources used. Look at the resources that were created either in the AWS Management Console or with the `list-stack-resources` AWS CLI command. You can also see resources configured to be deployed in the template itself.
- Determine all the services your workload requires by looking at the deployment code.
- Determine the service quotas that apply. Use the programmatically accessible information from Trusted Advisor and Service Quotas.

- Establish an automated monitoring method (see REL01-BP02 Manage service quotas across accounts and regions and REL01-BP04 Monitor and manage quotas) to alert and inform if services quotas are near or have reached their limit.
- Establish an automated and programmatic method to check if a service quota has been changed in one region but not in other regions in the same account (see REL01-BP02 Manage service quotas across accounts and regions and REL01-BP04 Monitor and manage quotas).
- Automate scanning application logs and metrics to determine if there are any quota or service constraint errors. If these errors are present, send alerts to the monitoring system.
- Establish engineering procedures to calculate the required change in quota (see REL01-BP05 Automate quota management) once it has been identified that larger quotas are required for specific services.
- Create a provisioning and approval workflow to request changes in service quota. This should include an exception workflow in case of request deny or partial approval.
- Create an engineering method to review service quotas prior to provisioning and using new AWS services before rolling out to production or loaded environments. (for example, load testing account).

For service constraints:

- Establish monitoring and metrics methods to alert for resources reading close to their resource constraints. Leverage CloudWatch as appropriate for metrics or log monitoring.
- Establish alert thresholds for each resource that has a constraint that is meaningful to the application or system.
- Create workflow and infrastructure management procedures to change the resource type if the constraint is near utilization. This workflow should include load testing as a best practice to verify that new type is the correct resource type with the new constraints.
- Migrate identified resource to the recommended new resource type, using existing procedures and processes.

Resources

Related best practices:

- REL01-BP02 Manage service quotas across accounts and regions
- REL01-BP03 Accommodate fixed service quotas and constraints through architecture

- REL01-BP04 Monitor and manage quotas
- REL01-BP05 Automate quota management
- REL01-BP06 Ensure that a sufficient gap exists between the current quotas and the maximum usage to accommodate failover
- REL03-BP01 Choose how to segment your workload
- REL10-BP01 Deploy the workload to multiple locations
- REL11-BP01 Monitor all components of the workload to detect failures
- REL11-BP03 Automate healing on all layers
- REL12-BP04 Test resiliency using chaos engineering

Related documents:

- AWS Well-Architected Framework's Reliability Pillar: Availability
- 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?
- How to Request Quota Increase
- Service endpoints and quotas
- Service Quotas User Guide
- Quota Monitor for AWS
- AWS Fault Isolation Boundaries
- Availability with redundancy
- AWS for Data
- What is Continuous Integration?
- What is Continuous Delivery?
- APN Partner: partners that can help with configuration management
- Managing the account lifecycle in account-per-tenant SaaS environments on AWS
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations

- Automating Service Limit Increases and Enterprise Support with AWS Control Tower

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas
- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo

Related tools:

- Amazon CodeGuru Reviewer
- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge
- Amazon DevOps Guru
- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager
- AWS Marketplace

REL01-BP02 Manage service quotas across accounts and regions

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

Desired outcome: Services and applications should not be affected by service quota exhaustion for configurations that span accounts or Regions or that have resilience designs using zone, Region, or account failover.

Common anti-patterns:

- Allowing resource usage in one isolation Region to grow with no mechanism to maintain capacity in the other ones.

- Manually setting all quotas independently in isolation Regions.
- Not considering the effect of resiliency architectures (like active or passive) in future quota needs during a degradation in the non-primary Region.
- Not evaluating quotas regularly and making necessary changes in every Region and account the workload runs.
- Not leveraging quota request templates to request increases across multiple Regions and accounts.
- Not updating service quotas due to incorrectly thinking that increasing quotas has cost implications like compute reservation requests.

Benefits of establishing this best practice: Verifying that you can handle your current load in secondary regions or accounts if regional services become unavailable. This can help reduce the number of errors or levels of degradations that occur during region loss.

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

Implementation guidance

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. Maintaining a high degree of resiliency requires that service quotas are assessed continually (whether automated or manual).

With more workloads spanning Regions due to the implementation of designs using *Active/Active*, *Active/Passive – Hot*, *Active/Passive-Cold*, and *Active/Passive-Pilot Light* approaches, it is essential to understand all Region and account quota levels. Past traffic patterns are not always a good indicator if the service quota is set correctly.

Equally important, the service quota name limit is not always the same for every Region. In one Region, the value could be five, and in another region the value could be ten. Management of these quotas must span all the same services, accounts, and Regions to provide consistent resilience under load.

Reconcile all the service quota differences across different Regions (Active Region or Passive Region) and create processes to continually reconcile these differences. The testing plans of passive Region failovers are rarely scaled to peak active capacity, meaning that game day or table top exercises can fail to find differences in service quotas between Regions and also then maintain the correct limits.

Service quota drift, the condition where service quota limits for a specific named quota is changed in one Region and not all Regions, is very important to track and assess. Changing the quota in Regions with traffic or potentially could carry traffic should be considered.

- 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. These values should be compared for differences.

Implementation steps

- Review Service Quotas values that might have breached beyond the a risk level of usage. AWS Trusted Advisor provides alerts for 80% and 90% threshold breaches.
- Review values for service quotas in any Passive Regions (in an Active/Passive design). Verify that load will successfully run in secondary Regions in the event of a failure in the primary Region.
- Automate assessing if any service quota drift has occurred between Regions in the same account and act accordingly to change the limits.
- If the customer Organizational Units (OU) are structured in the supported manner, service quota templates should be updated to reflect changes in any quotas that should be applied to multiple Regions and accounts.
  - Create a template and associate Regions to the quota change.
  - Review all existing service quota templates for any changes required (Region, limits, and accounts).

Resources

Related best practices:

- REL01-BP01 Aware of service quotas and constraints
- REL01-BP03 Accommodate fixed service quotas and constraints through architecture
- REL01-BP04 Monitor and manage quotas
- REL01-BP05 Automate quota management
- REL01-BP06 Ensure that a sufficient gap exists between the current quotas and the maximum usage to accommodate failover
- REL03-BP01 Choose how to segment your workload

- REL10-BP01 Deploy the workload to multiple locations
- REL11-BP01 Monitor all components of the workload to detect failures
- REL11-BP03 Automate healing on all layers
- REL12-BP04 Test resiliency using chaos engineering

Related documents:

- AWS Well-Architected Framework's Reliability Pillar: Availability
- 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?
- How to Request Quota Increase
- Service endpoints and quotas
- Service Quotas User Guide
- Quota Monitor for AWS
- AWS Fault Isolation Boundaries
- Availability with redundancy
- AWS for Data
- What is Continuous Integration?
- What is Continuous Delivery?
- APN Partner: partners that can help with configuration management
- Managing the account lifecycle in account-per-tenant SaaS environments on AWS
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas

- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo

Related services:

- Amazon CodeGuru Reviewer
- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge
- Amazon DevOps Guru
- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager
- AWS Marketplace

REL01-BP03 Accommodate fixed service quotas and constraints through architecture

Be aware of unchangeable service quotas, service constraints, and physical resource limits. Design architectures for applications and services to prevent these limits from impacting reliability.

Examples include network bandwidth, serverless function invocation payload size, throttle burst rate for of an API gateway, and concurrent user connections to a database.

Desired outcome: The application or service performs as expected under normal and high traffic conditions. They have been designed to work within the limitations for that resource's fixed constraints or service quotas.

Common anti-patterns:

- Choosing a design that uses a resource of a service, unaware that there are design constraints that will cause this design to fail as you scale.

- Performing benchmarking that is unrealistic and will reach service fixed quotas during the testing. For example, running tests at a burst limit but for an extended amount of time.
- Choosing a design that cannot scale or be modified if fixed service quotas are to be exceeded. For example, an SQS payload size of 256KB.
- Observability has not been designed and implemented to monitor and alert on thresholds for service quotas that might be at risk during high traffic events

Benefits of establishing this best practice: Verifying that the application will run under all projected services load levels without disruption or degradation.

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

Implementation guidance

Unlike soft service quotas or resources that be replaced with higher capacity units, AWS services' fixed quotas cannot be changed. This means that all these type of AWS services must be evaluated for potential hard capacity limits when used in an application design.

Hard limits are shown in the Service Quotas console. If the columns shows `ADJUSTABLE = No`, the service has a hard limit. Hard limits are also shown in some resources configuration pages. For example, Lambda has specific hard limits that cannot be adjusted.

As an example, when designing a python application to run in a Lambda function, the application should be evaluated to determine if there is any chance of Lambda running longer than 15 minutes. If the code may run more than this service quota limit, alternate technologies or designs must be considered. If this limit is reached after production deployment, the application will suffer degradation and disruption until it can be remediated. Unlike soft quotas, there is no method to change to these limits even under emergency Severity 1 events.

Once the application has been deployed to a testing environment, strategies should be used to find if any hard limits can be reached. Stress testing, load testing, and chaos testing should be part of the introduction test plan.

Implementation steps

- Review the complete list of AWS services that could be used in the application design phase.
- Review the soft quota limits and hard quota limits for all these services. Not all limits are shown in the Service Quotas console. Some services describe these limits in alternate locations.

- As you design your application, review your workload's business and technology drivers, such as business outcomes, use case, dependent systems, availability targets, and disaster recovery objects. Let your business and technology drivers guide the process to identify the distributed system that is right for your workload.
- Analyze service load across Regions and accounts. Many hard limits are regionally based for services. However, some limits are account based.
- Analyze resilience architectures for resource usage during a zonal failure and Regional failure. In the progression of multi-Region designs using active/active, active/passive – hot, active/passive - cold, and active/passive - pilot light approaches, these failure cases will cause higher usage. This creates a potential use case for hitting hard limits.

Resources

Related best practices:

- REL01-BP01 Aware of service quotas and constraints
- REL01-BP02 Manage service quotas across accounts and regions
- REL01-BP04 Monitor and manage quotas
- REL01-BP05 Automate quota management
- REL01-BP06 Ensure that a sufficient gap exists between the current quotas and the maximum usage to accommodate failover
- REL03-BP01 Choose how to segment your workload
- REL10-BP01 Deploy the workload to multiple locations
- REL11-BP01 Monitor all components of the workload to detect failures
- REL11-BP03 Automate healing on all layers
- REL12-BP04 Test resiliency using chaos engineering

Related documents:

- AWS Well-Architected Framework's Reliability Pillar: Availability
- 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?
- How to Request Quota Increase
- Service endpoints and quotas
- Service Quotas User Guide
- Quota Monitor for AWS
- AWS Fault Isolation Boundaries
- Availability with redundancy
- AWS for Data
- What is Continuous Integration?
- What is Continuous Delivery?
- APN Partner: partners that can help with configuration management
- Managing the account lifecycle in account-per-tenant SaaS environments on AWS
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower
- Actions, resources, and condition keys for Service Quotas

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas
- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo
- AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small

Related tools:

- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge

- Amazon DevOps Guru
- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager
- AWS Marketplace

REL01-BP04 Monitor and manage quotas

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

Desired outcome: Active and automated systems that manage and monitor have been deployed. These operations solutions ensure that quota usage thresholds are nearing being reached. These would be proactively remediated by requested quota changes.

Common anti-patterns:

- Not configuring monitoring to check for service quota thresholds
- Not configuring monitoring for hard limits, even though those values cannot be changed.
- Assuming that amount of time required to request and secure a soft quota change is immediate or a short period.
- 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 AWS Service Quotas and not monitoring other AWS services.
- Not considering quota management for multiple Region resiliency designs, like active/active, active/passive – hot, active/passive - cold, and active/passive - pilot light approaches.
- Not assessing quota differences between Regions.
- Not assessing the needs in every Region for a specific quota increase request.
- Not leveraging templates for multi-Region quota management.

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 help limit any degradations due to quota exhaustion.

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

Implementation guidance

For supported services, you can monitor your quotas by configuring various different services that can assess and then send alerts or alarms. This can aid in monitoring usage and can alert you to approaching quotas. These alarms can be invoked from AWS Config, Lambda functions, Amazon CloudWatch, or from AWS 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.

Implementation steps

For monitoring:

- Capture current resource consumption (for example, buckets or instances). Use service API operations, such as the Amazon EC2 `DescribeInstances` API, to collect current resource consumption.
- Capture your current quotas that are essential and applicable to the services using:
  - AWS Service Quotas
  - AWS Trusted Advisor
  - AWS documentation
  - AWS service-specific pages
  - AWS Command Line Interface (AWS CLI)
  - AWS Cloud Development Kit (AWS CDK)
- Use AWS Service Quotas, an AWS service that helps you manage your quotas for over 250 AWS services from one location.
- Use Trusted Advisor service limits to monitor your current service limits at various thresholds.
- Use the service quota history (console or AWS CLI) to check on regional increases.
- Compare service quota changes in each Region and each account to create equivalency, if required.

For management:

- Automated: Set up an AWS Config custom rule to scan service quotas across Regions and compare for differences.
- Automated: Set up a scheduled Lambda function to scan service quotas across Regions and compare for differences.
- Manual: Scan services quota through AWS CLI, API, or AWS Console to scan service quotas across Regions and compare for differences. Report the differences.
- If differences in quotas are identified between Regions, request a quota change, if required.
- Review the result of all requests.

Resources

Related best practices:

- REL01-BP01 Aware of service quotas and constraints
- REL01-BP02 Manage service quotas across accounts and regions
- REL01-BP03 Accommodate fixed service quotas and constraints through architecture
- REL01-BP05 Automate quota management
- REL01-BP06 Ensure that a sufficient gap exists between the current quotas and the maximum usage to accommodate failover
- REL03-BP01 Choose how to segment your workload
- REL10-BP01 Deploy the workload to multiple locations
- REL11-BP01 Monitor all components of the workload to detect failures
- REL11-BP03 Automate healing on all layers
- REL12-BP04 Test resiliency using chaos engineering

Related documents:

- AWS Well-Architected Framework's Reliability Pillar: Availability
- 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?

- How to Request Quota Increase
- Service endpoints and quotas
- Service Quotas User Guide
- Quota Monitor for AWS
- AWS Fault Isolation Boundaries
- Availability with redundancy
- AWS for Data
- What is Continuous Integration?
- What is Continuous Delivery?
- APN Partner: partners that can help with configuration management
- Managing the account lifecycle in account-per-tenant SaaS environments on AWS
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower
- Actions, resources, and condition keys for Service Quotas

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas
- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo
- AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small

Related tools:

- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge
- Amazon DevOps Guru

- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager
- AWS Marketplace

REL01-BP05 Automate quota management

Service quotas, also referred to as limits in AWS services, are the maximum values for the resources in your AWS account. Each AWS service defines a set of quotas and their default values. To provide your workload access to all the resources it needs, you might need to increase your service quota values.

Growth in workload consumption of AWS resources can threaten workload stability and impact the user experience if quotas are exceeded. Implement tools to alert you when your workload approaches the limits and consider creating quota increase requests automatically.

Desired outcome: Quotas are appropriately configured for the workloads running in each AWS account and Region.

Common anti-patterns:

- You fail to consider and adjust quotas appropriately to meet workload requirements.
- You track quotas and usage using methods that can become outdated, such as with spreadsheets.
- You only update service limits on periodic schedules.
- Your organization lacks operational processes to review existing quotas and request service quota increases when necessary.

Benefits of establishing this best practice:

- Enhanced workload resiliency: You prevent errors caused by exceeding AWS resource quotas.
- Simplified disaster recovery: You can reuse automated quota management mechanisms built in the primary Region during DR setup in another AWS Region.

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

Implementation guidance

View current quotas and track ongoing quota consumption through mechanisms such as AWS Service Quotas console, AWS Command Line Interface (AWS CLI), and AWS SDKs. You can also integrate your configuration management databases (CMDB) and IT service management (ITSM) systems with the AWS Service Quota APIs.

Generate automated alerts if quota usage reaches your defined thresholds, and define a process for submitting quota increase requests when you receive alerts. If the underlying workload is critical to your business, you can automate quota increase requests, but carefully test the automation to avoid the risk of runaway action such as a growth feedback loop.

Smaller quota increases are often automatically approved. Larger quota requests may need to be manually processed by AWS support and can take additional time to review and process. Allow for additional time to process multiple requests or large increase requests.

Implementation steps

- Implement automated monitoring of service quotas, and issue alerts if your workload's resource utilization growth approaches your quota limits. For example, Quota Monitor for AWS can provide automated monitoring of service quotas. This tool integrates with AWS Organizations and deploys using Cloudformation StackSets so that new accounts are automatically monitored on creation.
- Use features such as Service Quotas request templates or AWS Control Tower to simplify Service Quotas setup for new accounts.
- Build dashboards of your current service quota use across all AWS accounts and regions and reference them as necessary to prevent exceeding your quotas. Trusted Advisor Organizational (TAO) Dashboard, part of the Cloud Intelligence Dashboards, can get you quickly started with such a dashboard.
- Track service limit increase requests. Consolidated Insights from Multiple Accounts(CIMA) can provide an Organization-level view of all your requests.
- Test alert generation and any quota increase request automation by setting lower quota thresholds in non-production accounts. Do not conduct these tests in a production account.

Resources

Related best practices:

- OPS10-BP07 Automate responses to events

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 Solution on AWS - AWS Solution
- What is Service Quotas?
- What is Service Quotas request templates?

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower

Related tools:

- Quota Monitor for AWS

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

This article explains how to maintain space between the resource quota and your usage, and how it can benefit your organization. After you finish using a resource, the usage quota may continue to account for that resource. This can result in a failing or inaccessible resource. Prevent resource failure by verifying that your quotas cover the overlap of inaccessible resources and their replacements. Consider cases like network failure, Availability Zone failure, or Region failures when calculating this gap.

Desired outcome: Small or large failures in resources or resource accessibility can be covered within the current service thresholds. Zone failures, network failures, or even Regional failures have been considered in the resource planning.

Common anti-patterns:

- Setting service quotas based on current needs without accounting for failover scenarios.
- Not considering the principals of static stability when calculating the peak quota for a service.
- Not considering the potential of inaccessible resources in calculating total quota needed for each Region.
- Not considering AWS service fault isolation boundaries for some services and their potential abnormal usage patterns.

Benefits of establishing this best practice: When service disruption events impact application availability, use the cloud to implement strategies to recover from these events. An example strategy is creating additional resources to replace inaccessible resources to accommodate failover conditions without exhausting your service limit.

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

Implementation guidance

When evaluating a quota limit, consider failover cases that might occur due to some degradation. Consider the following failover cases.

- A disrupted or inaccessible VPC.
- An inaccessible subnet.
- A degraded Availability Zone that impacts resource accessibility.
- Networking routes or ingress and egress points are blocked or changed.
- A degraded Region that impacts resource accessibility.
- A subset of resources affected by a failure in a Region or an Availability Zone.

The decision to failover is unique for each situation, as the business impact can vary. Address resource capacity planning in the failover location and the resources' quotas before deciding to failover an application or service.

Consider higher than normal peaks of activity when reviewing quotas for each service. These peaks might be related to resources that are inaccessible due to networking or permissions, but are still active. Unterminated active resources count against the service quota limit.

Implementation steps

- Maintain space between your service quota and your maximum usage to accommodate for a failover or loss of accessibility.
- Determine your service quotas. Account for typical deployment patterns, availability requirements, and consumption growth.
- Request quota increases if necessary. Anticipate a wait time for the quota increase request.
- Determine your reliability requirements (also known as your number of nines).
- Understand potential fault scenarios such as loss of a component, an Availability Zone, or a Region.
- Establish your deployment methodology (examples include canary, blue/green, red/black, and rolling).
- Include an appropriate buffer to the current quota limit. An example buffer could be 15%.
- Include calculations for static stability (Zonal and Regional) where appropriate.
- Plan consumption growth and monitor your consumption trends.
- Consider the static stability impact for your most critical workloads. Assess resources conforming to a statically stable system in all Regions and Availability Zones.
- Consider using On-Demand Capacity Reservations to schedule capacity ahead of any failover. This is a useful strategy to implement for critical business schedules to reduce potential risks of obtaining the correct quantity and type of resources during failover.

Resources

Related best practices:

- REL01-BP01 Aware of service quotas and constraints
- REL01-BP02 Manage service quotas across accounts and regions
- REL01-BP03 Accommodate fixed service quotas and constraints through architecture
- REL01-BP04 Monitor and manage quotas
- REL01-BP05 Automate quota management
- REL03-BP01 Choose how to segment your workload
- REL10-BP01 Deploy the workload to multiple locations
- REL11-BP01 Monitor all components of the workload to detect failures
- REL11-BP03 Automate healing on all layers
- REL12-BP04 Test resiliency using chaos engineering

Related documents:

- AWS Well-Architected Framework's Reliability Pillar: Availability
- 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?
- How to Request Quota Increase
- Service endpoints and quotas
- Service Quotas User Guide
- Quota Monitor for AWS
- AWS Fault Isolation Boundaries
- Availability with redundancy
- AWS for Data
- What is Continuous Integration?
- What is Continuous Delivery?
- APN Partner: partners that can help with configuration management
- Managing the account lifecycle in account-per-tenant SaaS environments on AWS
- Managing and monitoring API throttling in your workloads
- View AWS Trusted Advisor recommendations at scale with AWS Organizations
- Automating Service Limit Increases and Enterprise Support with AWS Control Tower
- Actions, resources, and condition keys for Service Quotas

Related videos:

- AWS Live re:Inforce 2019 - Service Quotas
- View and Manage Quotas for AWS Services Using Service Quotas
- AWS IAM Quotas Demo
- AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small

Related tools:

- AWS CodeDeploy
- AWS CloudTrail
- Amazon CloudWatch
- Amazon EventBridge
- Amazon DevOps Guru
- AWS Config
- AWS Trusted Advisor
- AWS CDK
- AWS Systems Manager
- AWS Marketplace

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
- REL02-BP02 Provision redundant connectivity between private networks in the cloud and on-premises environments
- REL02-BP03 Ensure IP subnet allocation accounts for expansion and availability
- REL02-BP04 Prefer hub-and-spoke topologies over many-to-many mesh
- REL02-BP05 Enforce non-overlapping private IP address ranges in all private address spaces where they are connected

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

Building highly available network connectivity to public endpoints of your workloads can help you reduce downtime due to loss of connectivity and improve the availability and SLA of your workload. To achieve this, use highly available DNS, content delivery networks (CDNs), API gateways, load balancing, or reverse proxies.

Desired outcome: It is critical to plan, build, and operationalize highly available network connectivity for your public endpoints. If your workload becomes unreachable due to a loss in connectivity, even if your workload is running and available, your customers will see your system as down. By combining the highly available and resilient network connectivity for your workload's public endpoints, along with a resilient architecture for your workload itself, you can provide the best possible availability and service level for your customers.

AWS Global Accelerator, Amazon CloudFront, Amazon API Gateway, AWS Lambda Function URLs, AWS AppSync APIs, and Elastic Load Balancing (ELB) all provide highly available public endpoints. Amazon Route 53 provides a highly available DNS service for domain name resolution to verify that your public endpoint addresses can be resolved.

You can also evaluate AWS Marketplace software appliances for load balancing and proxying.

Common anti-patterns:

- Designing a highly available workload without planning out DNS and network connectivity for high availability.
- Using public internet addresses on individual instances or containers and managing the connectivity to them with DNS.
- Using IP addresses instead of domain names for locating services.
- Not testing out scenarios where connectivity to your public endpoints is lost.
- Not analyzing network throughput needs and distribution patterns.
- Not testing and planning for scenarios where internet network connectivity to your public endpoints of your workload might be interrupted.
- Providing content (like web pages, static assets, or media files) to a large geographic area and not using a content delivery network.
- Not planning for distributed denial of service (DDoS) attacks. DDoS attacks risk shutting out legitimate traffic and lowering availability for your users.

Benefits of establishing this best practice: Designing for highly available and resilient network connectivity ensures that your workload is accessible and available to your users.

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

Implementation guidance

At the core of building highly available network connectivity to your public endpoints is the routing of the traffic. To verify your traffic is able to reach the endpoints, the DNS must be able to resolve the domain names to their corresponding IP addresses. Use a highly available and scalable Domain Name System (DNS) such as Amazon Route 53 to manage your domain's DNS records. You can also use health checks provided by Amazon Route 53. The health checks verify that your application is reachable, available, and functional, and they can be set up in a way that they mimic your user's behavior, such as requesting a web page or a specific URL. In case of failure, Amazon Route 53 responds to DNS resolution requests and directs the traffic to only healthy endpoints. You can also consider using Geo DNS and Latency Based Routing capabilities offered by Amazon Route 53.

To verify that your workload itself is highly available, use Elastic Load Balancing (ELB). Amazon Route 53 can be used to target traffic to ELB, which distributes the traffic to the target compute instances. You can also use Amazon API Gateway along with AWS Lambda for a serverless solution. Customers can also run workloads in multiple AWS Regions. With multi-site active/active pattern, the workload can serve traffic from multiple Regions. With a multi-site active/passive pattern, the workload serves traffic from the active region while data is replicated to the secondary region and becomes active in the event of a failure in the primary region. Route 53 health checks can then be used to control DNS failover from any endpoint in a primary Region to an endpoint in a secondary Region, verifying that your workload is reachable and available to your users.

Amazon CloudFront provides a simple API for distributing content with low latency and high data transfer rates by serving requests using a network of edge locations around the world. Content delivery networks (CDNs) serve customers by serving content located or cached at a location near to the user. This also improves availability of your application as the load for content is shifted away from your servers over to CloudFront's edge locations. The edge locations and regional edge caches hold cached copies of your content close to your viewers resulting in quick retrieval and increasing reachability and availability of your workload.

For workloads with users spread out geographically, AWS Global Accelerator helps you improve the availability and performance of the applications. AWS Global Accelerator provides Anycast static IP addresses that serve as a fixed entry point to your application hosted in one or more

AWS Regions. This allows traffic to ingress onto the AWS global network as close to your users as possible, improving reachability and availability of your workload. AWS Global Accelerator also monitors the health of your application endpoints by using TCP, HTTP, and HTTPS health checks. Any changes in the health or configuration of your endpoints permit redirection of user traffic to healthy endpoints that deliver the best performance and availability to your users. In addition, AWS Global Accelerator has a fault-isolating design that uses two static IPv4 addresses that are serviced by independent network zones increasing the availability of your applications.

To help protect customers from DDoS attacks, AWS provides AWS Shield Standard. Shield Standard comes automatically turned on and protects from common infrastructure (layer 3 and 4) attacks like SYN/UDP floods and reflection attacks to support high availability of your applications on AWS. For additional protections against more sophisticated and larger attacks (like UDP floods), state exhaustion attacks (like TCP SYN floods), and to help protect your applications running on Amazon Elastic Compute Cloud (Amazon EC2), Elastic Load Balancing (ELB), Amazon CloudFront, AWS Global Accelerator, and Route 53, you can consider using AWS Shield Advanced. For protection against Application layer attacks like HTTP POST or GET floods, use AWS WAF. AWS WAF can use IP addresses, HTTP headers, HTTP body, URI strings, SQL injection, and cross-site scripting conditions to determine if a request should be blocked or allowed.

Implementation steps

1. Set up highly available DNS: Amazon Route 53 is a highly available and scalable domain name system (DNS) web service. Route 53 connects user requests to internet applications running on AWS or on-premises. For more information, see configuring Amazon Route 53 as your DNS service.
2. Setup health checks: When using Route 53, verify that only healthy targets are resolvable. Start by creating Route 53 health checks and configuring DNS failover. The following aspects are important to consider when setting up health checks:
   a. How Amazon Route 53 determines whether a health check is healthy
   b. Creating, updating, and deleting health checks
   c. Monitoring health check status and getting notifications
   d. Best practices for Amazon Route 53 DNS
3. Connect your DNS service to your endpoints.
   a. When using Elastic Load Balancing as a target for your traffic, create an alias record using Amazon Route 53 that points to your load balancer's regional endpoint. During the creation of the alias record, set the Evaluate target health option to Yes.

   b. For serverless workloads or private APIs when API Gateway is used, use Route 53 to direct traffic to API Gateway.

4. Decide on a content delivery network.

   a. For delivering content using edge locations closer to the user, start by understanding how CloudFront delivers content.

   b. Get started with a simple CloudFront distribution. CloudFront then knows where you want the content to be delivered from, and the details about how to track and manage content delivery. The following aspects are important to understand and consider when setting up CloudFront distribution:

      i. How caching works with CloudFront edge locations

      ii. Increasing the proportion of requests that are served directly from the CloudFront caches (cache hit ratio)

      iii. Using Amazon CloudFront Origin Shield

      iv. Optimizing high availability with CloudFront origin failover

5. Set up application layer protection: AWS WAF helps you protect against common web exploits and bots that can affect availability, compromise security, or consume excessive resources. To get a deeper understanding, review how AWS WAF works and when you are ready to implement protections from application layer HTTP POST AND GET floods, review Getting started with AWS WAF. You can also use AWS WAF with CloudFront see the documentation on how AWS WAF works with Amazon CloudFront features.

6. Set up additional DDoS protection: By default, all AWS customers receive protection from common, most frequently occurring network and transport layer DDoS attacks that target your web site or application with AWS Shield Standard at no additional charge. For additional protection of internet-facing applications running on Amazon EC2, Elastic Load Balancing, Amazon CloudFront, AWS Global Accelerator, and Amazon Route 53 you can consider AWS Shield Advanced and review examples of DDoS resilient architectures. To protect your workload and your public endpoints from DDoS attacks review Getting started with AWS Shield Advanced.

Resources

Related best practices:

- REL10-BP01 Deploy the workload to multiple locations
- REL11-BP04 Rely on the data plane and not the control plane during recovery

- REL11-BP06 Send notifications when events impact availability

Related documents:

- APN Partner: partners that can help plan your networking
- AWS Marketplace for Network Infrastructure
- What Is AWS Global Accelerator?
- What is Amazon CloudFront?
- What is Amazon Route 53?
- What is Elastic Load Balancing?
- Network Connectivity capability - Establishing Your Cloud Foundations
- What is Amazon API Gateway?
- What are AWS WAF, AWS Shield, and AWS Firewall Manager?
- What is Amazon Application Recovery Controller?
- Configure custom health checks for DNS failover

Related videos:

- AWS re:Invent 2022 - Improve performance and availability with AWS Global Accelerator
- AWS re:Invent 2020: Global traffic management with Amazon Route 53
- AWS re:Invent 2022 - Operating highly available Multi-AZ applications
- AWS re:Invent 2022 - Dive deep on AWS networking infrastructure
- AWS re:Invent 2022 - Building resilient networks

Related examples:

- Disaster Recovery with Amazon Application Recovery Controller (ARC)
- AWS Global Accelerator Workshop

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

Implement redundancy in your connections between private networks in the cloud and on-premises environments to achieve connectivity resilience. This can be accomplished by deploying two or more links and traffic paths, preserving connectivity in the event of network failures.

Common anti-patterns:

- You depend on just one network connection, which creates a single point of failure.
- You use only one VPN tunnel or multiple tunnels that end in the same Availability Zone.
- You rely on one ISP for VPN connectivity, which can lead to complete failures during ISP outages.
- Not implementing dynamic routing protocols like BGP, which are crucial for rerouting traffic during network disruptions.
- You ignore the bandwidth limitations of VPN tunnels and overestimate their backup capabilities.

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

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

Implementation guidance

When using AWS Direct Connect to connect your on-premises network to AWS, you can achieve maximum network resiliency (SLA of 99.99%) by using separate connections that end on distinct devices in more than one on-premises location and more than one AWS Direct Connect location. This topology offers resilience against device failures, connectivity issues, and complete location outages. Alternatively, you can achieve high resiliency (SLA of 99.9%) by using two individual connections to multiple locations (each on-premises location connected to a single Direct Connect location). This approach protects against connectivity disruptions caused by fiber cuts or device failures and helps mitigate complete location failures. The AWS Direct Connect Resiliency Toolkit can assist in designing your AWS Direct Connect topology.

You can also consider AWS Site-to-Site VPN ending on an AWS Transit Gateway as a cost-effective backup to your primary AWS Direct Connect connection. This setup enables equal-cost multipath (ECMP) routing across multiple VPN tunnels, allowing for throughput of up to 50Gbps, even

though each VPN tunnel is capped at 1.25 Gbps. It's important to note, however, that AWS Direct Connect is still the most effective choice for minimizing network disruptions and providing stable connectivity.

When using VPNs over the internet to connect your cloud environment to your on-premises data center, configure two VPN tunnels as part of a single site-to-site VPN connection. Each tunnel should end in a different Availability Zone for high availability and use redundant hardware to prevent on-premises device failure. Additionally, consider multiple internet connections from various internet service providers (ISPs) at your on-premises location to avoid complete VPN connectivity disruption due to a single ISP outage. Selecting ISPs with diverse routing and infrastructure, especially those with separate physical paths to AWS endpoints, provides high connectivity availability.

In addition to physical redundancy with multiple AWS Direct Connect connections and multiple VPN tunnels (or a combination of both), implementing Border Gateway Protocol (BGP) dynamic routing is also crucial. Dynamic BGP provides automatic rerouting of traffic from one path to another based on real-time network conditions and configured policies. This dynamic behavior is especially beneficial in maintaining network availability and service continuity in the event of link or network failures. It quickly selects alternative paths, enhancing the network's resilience and reliability.

Implementation steps

- Acquisition highly-available connectivity between AWS and your on-premises environment.
  - Use multiple AWS Direct Connect connections or VPN tunnels between separately deployed private networks.
  - Use multiple AWS Direct Connect locations for high availability.
  - If using multiple AWS Regions, create redundancy in at least two of them.
- Use AWS Transit Gateway, when possible, to end your VPN connection.
- Evaluate AWS Marketplace appliances to end VPNs or extend your SD-WAN to AWS. If you use AWS Marketplace appliances, deploy redundant instances for high availability in different Availability Zones.
- Provide a redundant connection to your on-premises environment.
  - You may need redundant connections to multiple AWS Regions to achieve your availability needs.
  - Use the AWS Direct Connect Resiliency Toolkit to get started.

Resources

Related documents:

- AWS Direct Connect Resiliency Recommendations
- Using Redundant Site-to-Site VPN Connections to Provide Failover
- Routing policies and BGP communities
- Active/Active and Active/Passive Configurations in AWS Direct Connect
- APN Partner: partners that can help plan your networking
- AWS Marketplace for Network Infrastructure
- Amazon Virtual Private Cloud Connectivity Options Whitepaper
- Building a Scalable and Secure Multi-VPC AWS Network Infrastructure
- Using redundant Site-to-Site VPN connections to provide failover
- Using the AWS 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
- AWS re:Invent 2019: AWS Transit Gateway reference architectures for many VPCs

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 so that you can cover multiple Availability Zones.
- Consider leaving 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 Amazon EC2 instances that you might use, such as Spot Fleets for machine learning, Amazon EMR clusters, or Amazon Redshift clusters. Similar consideration should be given to Kubernetes clusters, such as Amazon Elastic Kubernetes Service (Amazon EKS), as each Kubernetes pod is assigned a routable address from the VPC CIDR block by default.
- 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.
- 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.
- The largest possible VPC CIDR block is a /16, and the smallest is a /28.
- Consider other connected networks (VPC, on-premises, or other cloud providers) and ensure non-overlapping IP address space. For more information, see REL02-BP05 Enforce non-overlapping private IP address ranges in all private address spaces where they are connected.

Desired outcome: A scalable IP subnet can help you accomodate for future growth and avoid unnecessary waste.

Common anti-patterns:

- Failing to consider future growth, resulting in CIDR blocks that are too small and requiring reconfiguration, potentially causing downtime.
- Incorrectly estimating how many IP addresses an elastic load balancer can use.
- Deploying many high traffic load balancers into the same subnets
- Using automated scaling mechanisms whilst failing to monitor IP address consumption.
- Defining excessively large CIDR ranges well beyond future growth expectations, which can lead to difficulty peering with other networks with overlapping address ranges.

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 with appropriately-sized CIDR blocks to accommodate your current and future needs.
    - If you have unknown growth projections, you may wish to err on the side of larger CIDR blocks to reduce the potential for future reconfiguration
  - Consider using IPv6 addressing for subnets as part of a dual-stack VPC. IPv6 is well suited to being used in private subnets containing fleets of ephemeral instances or containers that would otherwise require large numbers of IPv4 addresses.

Resources

Related Well-Architected best practices:

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

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?
- IPv6 on AWS
- IPv6 on reference architectures
- Amazon Elastic Kubernetes Service launches IPv6 support
- Recommendations for your VPC - Classic Load Balancers
- Availability Zone subnets - Application Load Balancers
- Availability Zones - Network Load Balancers

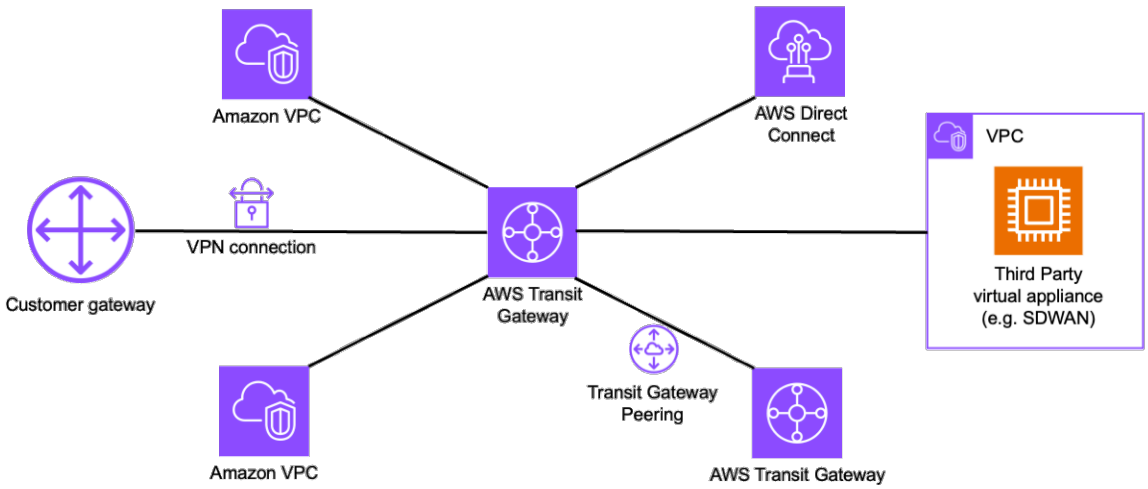
Related videos:

- 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)
- AWS re:Invent 2023: AWS Ready for what's next? Designing networks for growth and flexibility (NET310)

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

When connecting multiple private networks, such as Virtual Private Clouds (VPCs) and on-premises networks, opt for a hub-and-spoke topology over a meshed one. Unlike meshed topologies, where each network connects directly to the others and increases the complexity and management overhead, the hub-and-spoke architecture centralizes connections through a single hub. This centralization simplifies the network structure and enhances its operability, scalability, and control.

AWS Transit Gateway is a managed, scalable, and highly-available service designed for construction of hub-and-spoke networks on AWS. It serves as the central hub of your network that provides network segmentation, centralized routing, and the simplified connection to both cloud and on-premises environments. The following figure illustrates how you can use AWS Transit Gateway to build your hub-and-spoke topology.



Desired outcome: You have connected your Virtual Private Clouds (VPCs) and on-premises networks through a central hub. You configure your peering connections through the hub, which acts as a highly scalable cloud router. Routing is simplified because you do not have to work with complex peering relationships. Traffic between networks is encrypted, and you have the ability to isolate networks.

Common anti-patterns:

- You build complex network peering rules.
- You provide routes between networks that should not communicate with one another (for example, separate workloads that have no interdependencies).
- There is ineffective governance of the hub instance.

Benefits of establishing this best practice: As the number of connected networks increases, management and expansion of meshed connectivity becomes increasingly challenging. A mesh architecture introduces additional challenges, such as additional infrastructure components, configuration requirements, and deployment considerations. The mesh also introduces additional overhead to manage and monitor the data plane and control plane components. You must think

about how to provide high availability of the mesh architecture, how to monitor the mesh health and performance, and how to handle upgrades of the mesh components.

A hub-and-spoke model, on the other hand, establishes centralized traffic routing across multiple networks. It provides a simpler approach to management and monitoring of the data plane and control plane components.

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

Implementation guidance

Create a Network Services account if one does not exist. Place the hub in the organization's Network Services account. This approach allows the hub to be centrally managed by network engineers.

The hub of the hub-and-spoke model acts as a virtual router for traffic flowing between your Virtual Private Clouds (VPCs) and on-premises networks. This approach reduces network complexity and makes it easier to troubleshoot networking issues.

Consider your network design, including the VPCs, AWS Direct Connect, and Site-to-Site VPN connections you want to interconnect.

Consider using a separate subnet for each transit gateway VPC attachment. For each subnet, use a small CIDR (for example /28) so that you have more address space for compute resources. Additionally, create one network ACL, and associate it with all of the subnets that are associated with the hub. Keep the network ACL open in both the inbound and outbound directions.

Design and implement your routing tables such that routes are provided only between networks that should communicate. Omit routes between networks that should not communicate with one another (for example, between separate workloads that have no inter-dependencies).

Implementation steps

1. Plan your network. Determine which networks you want to connect, and verify that they don't share overlapping CIDR ranges.
2. Create an AWS Transit Gateway and attach your VPCs.
3. If needed, create VPN connections or Direct Connect gateways, and associate them with the Transit Gateway.
4. Define how traffic is routed between the connected VPCs and other connections through configuration of your Transit Gateway route tables.

5. Use Amazon CloudWatch to monitor and adjust configurations as necessary for performance and cost optimization.

Resources

Related best practices:

- REL02-BP03 Ensure IP subnet allocation accounts for expansion and availability
- REL02-BP05 Enforce non-overlapping private IP address ranges in all private address spaces where they are connected

Related documents:

- What Is a Transit Gateway?
- Transit gateway design best practices
- Building a Scalable and Secure Multi-VPC AWS Network Infrastructure
- Building a global network using AWS Transit Gateway Inter-Region peering
- Amazon Virtual Private Cloud Connectivity Options
- APN Partner: partners that can help plan your networking
- AWS Marketplace for Network Infrastructure

Related videos:

- AWS re:Invent 2023 - AWS networking foundations
- AWS re:Invent 2023 - Advanced VPC designs and new capabilities

Related workshops:

- AWS Transit Gateway Workshop

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, connected via Transit Gateway, or connected over VPN. 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 automating this.

Desired outcome:

- No IP address range conflicts between VPCs, on-premises environments, or other cloud providers.
- Proper IP address management allows for easier scaling of network infrastructure to accommodate growth and changes in network requirements.

Common anti-patterns:

- Using the same IP range in your VPC as you have on premises, in your corporate network, or other cloud providers
- Not tracking IP ranges of VPCs used to deploy your workloads.
- Relying on manual IP address management processes, such as spreadsheets.
- Over- or under-sizing CIDR blocks, which results in IP address waste or insufficient address space for your workload.

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

Make use of an IPAM, such as the Amazon VPC IP Address Manager, to monitor and manage your CIDR use. Several IPAMs are also available from the AWS Marketplace. Evaluate your potential usage on AWS, add CIDR ranges to existing VPCs, and create VPCs to allow planned growth in usage.

Implementation steps

- Capture current CIDR consumption (for example, VPCs and subnets).
  - Use service API operations to collect current CIDR consumption.
  - Use the Amazon VPC IP Address Manager to discover resources.

- Capture your current subnet usage.
  - Use service API operations to collect subnets per VPC in each Region.
  - Use the Amazon VPC IP Address Manager to discover resources.
- 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 consider using techniques like private NAT Gateway or AWS PrivateLink if you need to connect the overlapping ranges.

Resources

Related best practices:

- Protecting networks

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
- Connecting Networks with Overlapping IP Ranges
- What Is Amazon VPC?
- What is IPAM?

Related videos:

- AWS re:Invent 2023 - Advanced VPC designs and new capabilities
- AWS re:Invent 2019: AWS Transit Gateway reference architectures for many VPCs
- AWS re:Invent 2023 - Ready for what's next? Designing networks for growth and flexibility
- AWS re:Invent 2021 - {New Launch} Manage your IP addresses at scale on AWS

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

- REL03-BP02 Build services focused on specific business domains and functionality
- REL03-BP03 Provide service contracts per API

REL03-BP01 Choose how to segment your workload

Workload segmentation is important when determining the resilience requirements of your application. Monolithic architecture should be avoided whenever possible. Instead, carefully consider which application components can be broken out into microservices. Depending on your application requirements, this may end up being a combination of a service-oriented architecture (SOA) with microservices where possible. Workloads that are capable of statelessness are more capable of being deployed as microservices.

Desired outcome: Workloads should be supportable, scalable, and as loosely coupled as possible.

When making choices about how to segment your workload, 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. When refactoring an existing monolith, you will need to consider how well the application will support a decomposition towards statelessness. Breaking services into smaller pieces allows small, well-defined teams to develop and manage them. However, smaller services can introduce complexities which include possible increased latency, more complex debugging, and increased operational burden.

Common anti-patterns:

- The microservice *Death Star* is a situation in which the atomic components become so highly interdependent that a failure of one results in a much larger failure, making the components as rigid and fragile as a monolith.

Benefits of establishing this practice:

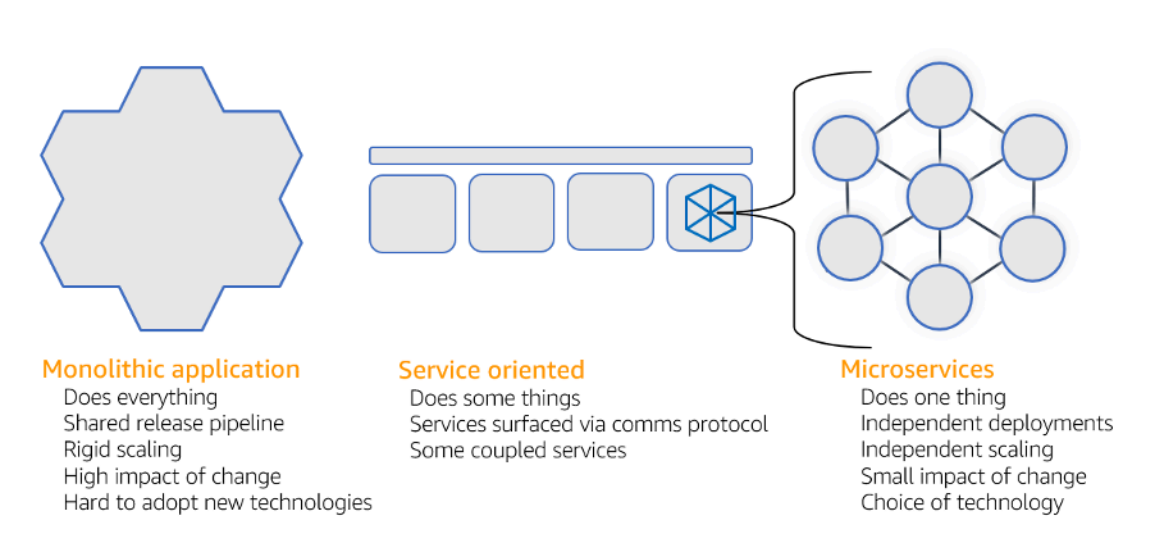
- More specific segments lead to greater agility, organizational flexibility, and scalability.
- Reduced impact of service interruptions.
- Application components may have different availability requirements, which can be supported by a more atomic segmentation.
- Well-defined responsibilities for teams supporting the workload.

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 an SOA or microservices architecture (or in some rare cases, a monolithic architecture). 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 primary trade-off 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 the debugging and tracing of user interactions. You can use AWS X-Ray to assist you in solving this problem. Another effect to consider is increased operational complexity as you increase the number of applications that you are managing, which requires the deployment of multiple independency components.



Monolithic, service-oriented, and microservices architectures

Implementation steps

- Determine the appropriate architecture to refactor or build your application. SOA and microservices offer respectively smaller segmentation, which is preferred as a modern scalable and reliable architecture. SOA can be a good compromise for achieving smaller segmentation while avoiding some of the complexities of microservices. For more details, see 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. For more details, see Implementing Microservices on AWS.
- Consider following the *Strangler Fig* pattern to refactor a monolith into smaller components. This involves gradually replacing specific application components with new applications and services. AWS Migration Hub Refactor Spaces acts as the starting point for incremental refactoring. For more details, see Seamlessly migrate on-premises legacy workloads using a strangler pattern.
- Implementing microservices may require a service discovery mechanism to allow these distributed services to communicate with each other. AWS App Mesh can be used with service-oriented architectures to provide reliable discovery and access of services. AWS Cloud Map can also be used for dynamic, DNS-based service discovery.
- If you're migrating from a monolith to SOA, Amazon MQ can help bridge the gap as a service bus when redesigning legacy applications in the cloud.
- For existing monoliths with a single, shared database, choose how to reorganize the data into smaller segments. This could be by business unit, access pattern, or data structure. At this point in the refactoring process, you should choose to move forward with a relational or non-relational (NoSQL) type of database. For more details, see From SQL to NoSQL.

Level of effort for the implementation plan: High

Resources

Related best practices:

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

Related documents:

- Amazon API Gateway: Configuring a REST API Using OpenAPI
- What is Service-Oriented Architecture?
- 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?

Related examples:

- Iterative App Modernization Workshop

Related videos:

- Delivering Excellence with Microservices on AWS

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

Service-oriented architectures (SOA) define services with well-delineated functions defined by business needs. Microservices use domain models and bounded context to draw service boundaries along business context boundaries. Focusing on business domains and functionality helps teams define independent reliability requirements for their services. Bounded contexts isolate and encapsulate business logic, allowing teams to better reason about how to handle failures.

Desired outcome: Engineers and business stakeholders jointly define bounded contexts and use them to design systems as services that fulfill specific business functions. These teams use established practices like event storming to define requirements. New applications are designed as services well-defined boundaries and loosely coupling. Existing monoliths are decomposed into bounded contexts and system designs move towards SOA or microservice architectures. When monoliths are refactored, established approaches like bubble contexts and monolith decomposition patterns are applied.

Domain-oriented services are executed as one or more processes that don't share state. They independently respond to fluctuations in demand and handle fault scenarios in light of domain specific requirements.

Common anti-patterns:

- Teams are formed around specific technical domains like UI and UX, middleware, or database instead of specific business domains.

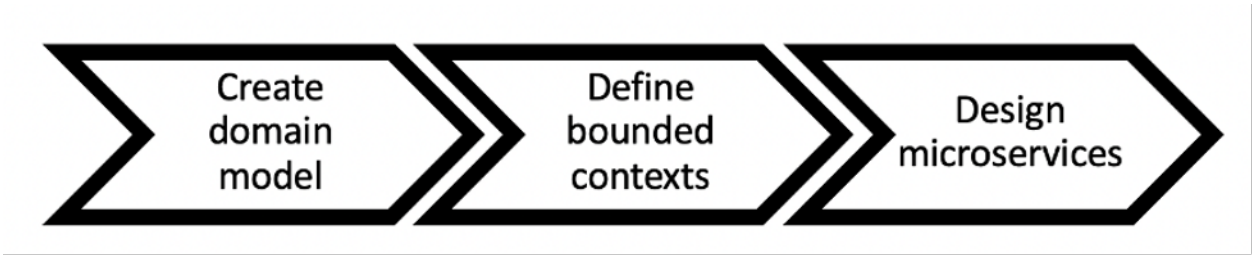
- Applications span domain responsibilities. Services that span bounded contexts can be more difficult to maintain, require larger testing efforts, and require multiple domain teams to participate in software updates.
- Domain dependencies, like domain entity libraries, are shared across services such that changes for one service domain require changes to other service domains
- Service contracts and business logic don't express entities in a common and consistent domain language, resulting in translation layers that complicate systems and increase debugging efforts.

Benefits of establishing this best practice: Applications are designed as independent services bounded by business domains and use a common business language. Services are independently testable and deployable. Services meet domain specific resiliency requirements for the domain implemented.

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

Implementation guidance

Domain-driven design (DDD) is the foundational approach of designing and building software around business domains. It's helpful to work with an existing framework when building services focused on business domains. When working with existing monolithic applications, you can take advantage of decomposition patterns that provide established techniques to modernize applications into services.



Domain-driven design

Implementation steps

- Teams can hold event storming workshops to quickly identify events, commands, aggregates and domains in a lightweight sticky note format.
- Once domain entities and functions have been formed in a domain context, you can divide your domain into services using bounded context, where entities that share similar features and

attributes are grouped together. With the model divided into contexts, a template for how to boundary microservices emerges.

  - For example, the Amazon.com website entities might include package, delivery, schedule, price, discount, and currency.
  - Package, delivery, and schedule are grouped into the shipping context, while price, discount, and currency are grouped into the pricing context.
- Decomposing monoliths into microservices outlines patterns for refactoring microservices. Using patterns for decomposition by business capability, subdomain, or transaction aligns well with domain-driven approaches.
- Tactical techniques such as the bubble context allow you to introduce DDD in existing or legacy applications without up-front rewrites and full commitments to DDD. In a bubble context approach, a small bounded context is established using a service mapping and coordination, or anti-corruption layer, which protects the newly defined domain model from external influences.

After teams have performed domain analysis and defined entities and service contracts, they can take advantage of AWS services to implement their domain-driven design as cloud-based services.

- Start your development by defining tests that exercise business rules of your domain. Test-driven development (TDD) and behavior-driven development (BDD) help teams keep services focused on solving business problems.
- Select the AWS services that best meet your business domain requirements and microservice architecture:
  - AWS Serverless allows your team focus on specific domain logic instead of managing servers and infrastructure.
  - Containers at AWS simplify the management of your infrastructure, so you can focus on your domain requirements.
  - Purpose built databases help you match your domain requirements to the best fit database type.
- Building hexagonal architectures on AWS outlines a framework to build business logic into services working backwards from a business domain to fulfill functional requirements and then attach integration adapters. Patterns that separate interface details from business logic with AWS services help teams focus on domain functionality and improve software quality.

Resources

Related best practices:

- REL03-BP01 Choose how to segment your workload
- REL03-BP03 Provide service contracts per API

Related documents:

- AWS Microservices
- Implementing Microservices on AWS
- How to break a Monolith into Microservices
- Getting Started with DDD when Surrounded by Legacy Systems
- Domain-Driven Design: Tackling Complexity in the Heart of Software
- Building hexagonal architectures on AWS
- Decomposing monoliths into microservices
- Event Storming
- Messages Between Bounded Contexts
- Microservices
- Test-driven development
- Behavior-driven development

Related examples:

- Designing Cloud Native Microservices on AWS (from DDD/EventStormingWorkshop)

Related tools:

- AWS Cloud Databases
- Serverless on AWS
- Containers at AWS

REL03-BP03 Provide service contracts per API

Service contracts are documented agreements between API producers and consumers defined in a machine-readable API definition. A contract versioning strategy allows consumers to continue using the existing API and migrate their applications to a newer API when they are ready. Producer deployment can happen any time as long as the contract is followed. Service teams can use the technology stack of their choice to satisfy the API contract.

Desired outcome: Applications built with service-oriented or microservice architectures are able to operate independently while having integrated runtime dependency. Changes deployed to an API consumer or producer do not interrupt the stability of the overall system when both sides follow a common API contract. Components that communicate over service APIs can perform independent functional releases, upgrades to runtime dependencies, or fail over to a disaster recovery (DR) site with little or no impact to each other. In addition, discrete services are able to independently scale absorbing resource demand without requiring other services to scale in unison.

Common anti-patterns:

- Creating service APIs without strongly typed schemas. This results in APIs that cannot be used to generate API bindings and payloads that can't be programmatically validated.
- Not adopting a versioning strategy, which forces API consumers to update and release or fail when service contracts evolve.
- Error messages that leak details of the underlying service implementation rather than describe integration failures in the domain context and language.
- Not using API contracts to develop test cases and mock API implementations to allow for independent testing of service components.

Benefits of establishing this best practice: Distributed systems composed of components that communicate over API service contracts can improve reliability. Developers can catch potential issues early in the development process with type checking during compilation to verify that requests and responses follow the API contract and required fields are present. API contracts provide a clear self-documenting interface for APIs and provider better interoperability between different systems and programming languages.

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

Implementation guidance

Once you have identified business domains and determined your workload segmentation, you can develop your service APIs. First, define machine-readable service contracts for APIs, and then implement an API versioning strategy. When you are ready to integrate services over common protocols like REST, GraphQL, or asynchronous events, you can incorporate AWS services into your architecture to integrate your components with strongly-typed API contracts.

AWS services for service API contrats

Incorporate AWS services including Amazon API Gateway, AWS AppSync, and Amazon EventBridge into your architecture to use API service contracts in your application. Amazon API Gateway helps you integrate with directly native AWS services and other web services. API Gateway supports the OpenAPI specification and versioning. AWS AppSync is a managed GraphQL endpoint you configure by defining a GraphQL schema to define a service interface for queries, mutations and subscriptions. Amazon EventBridge uses event schemas to define events and generate code bindings for your events.

Implementation steps

- First, define a contract for your API. A contract will express the capabilities of an API as well as define strongly typed data objects and fields for the API input and output.
- When you configure APIs in API Gateway, you can import and export OpenAPI Specifications for your endpoints.
  - Importing an OpenAPI definition simplifies the creation of your API and can be integrated with AWS infrastructure as code tools like the AWS Serverless Application Model and AWS Cloud Development Kit (AWS CDK).
  - Exporting an API definition simplifies integrating with API testing tools and provides services consumer an integration specification.
- You can define and manage GraphQL APIs with AWS AppSync by defining a GraphQL schema file to generate your contract interface and simplify interaction with complex REST models, multiple database tables or legacy services.
- AWS Amplify projects that are integrated with AWS AppSync generate strongly typed JavaScript query files for use in your application as well as an AWS AppSync GraphQL client library for Amazon DynamoDB tables.
- When you consume service events from Amazon EventBridge, events adhere to schemas that already exist in the schema registry or that you define with the OpenAPI Spec. With a schema

defined in the registry, you can also generate client bindings from the schema contract to integrate your code with events.

- Extending or version your API. Extending an API is a simpler option when adding fields that can be configured with optional fields or default values for required fields.
  - JSON based contracts for protocols like REST and GraphQL can be a good fit for contract extension.
  - XML based contracts for protocols like SOAP should be tested with service consumers to determine the feasibility of contract extension.
- When versioning an API, consider implementing proxy versioning where a facade is used to support versions so that logic can be maintained in a single codebase.
  - With API Gateway you can use request and response mappings to simplify absorbing contract changes by establishing a facade to provide default values for new fields or to strip removed fields from a request or response. With this approach the underlying service can maintain a single codebase.

Resources

Related best practices:

- REL03-BP01 Choose how to segment your workload
- REL03-BP02 Build services focused on specific business domains and functionality
- REL04-BP02 Implement loosely coupled dependencies
- REL05-BP03 Control and limit retry calls
- REL05-BP05 Set client timeouts

Related documents:

- What Is An API (Application Programming Interface)?
- Implementing Microservices on AWS
- Microservice Trade-Offs
- Microservices - a definition of this new architectural term
- Microservices on AWS
- Working with API Gateway extensions to OpenAPI
- OpenAPI-Specification

- GraphQL: Schemas and Types
- Amazon EventBridge code bindings

Related examples:

- Amazon API Gateway: Configuring a REST API Using OpenAPI
- Amazon API Gateway to Amazon DynamoDB CRUD application using OpenAPI
- Modern application integration patterns in a serverless age: API Gateway Service Integration
- Implementing header-based API Gateway versioning with Amazon CloudFront
- AWS AppSync: Building a client application

Related videos:

- Using OpenAPI in AWS SAM to manage API Gateway

Related tools:

- Amazon API Gateway
- AWS AppSync
- Amazon EventBridge

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 the kind of distributed systems you depend on
- REL04-BP02 Implement loosely coupled dependencies
- REL04-BP03 Do constant work
- REL04-BP04 Make mutating operations idempotent

REL04-BP01 Identify the kind of distributed systems you depend on

Distributed systems can be synchronous, asynchronous, or batch. Synchronous systems must process requests as quickly as possible and communicate with each other by making synchronous request and response calls using HTTP/S, REST, or remote procedure call (RPC) protocols. Asynchronous systems communicate with each other by exchanging data asynchronously through an intermediary service without coupling individual systems. Batch systems receive a large volume of input data, run automated data processes without human intervention, and generate output data.

Desired outcome: Design a workload that effectively interacts with synchronous, asynchronous, and batch dependencies.

Common anti-patterns:

- Workload waits indefinitely for a response from its dependencies, which could lead to workload clients timing out, not knowing if their request has been received.
- Workload uses a chain of dependent systems that call each other synchronously. This requires each system to be available and to successfully process a request before the whole chain can succeed, leading to potentially brittle behavior and overall availability.
- Workload communicates with its dependencies asynchronously and rely on the concept of exactly-once guaranteed delivery of messages, when often it is still possible to receive duplicate messages.
- Workload does not use proper batch scheduling tools and allows concurrent execution of the same batch job.

Benefits of establishing this best practice: It is common for a given workload to implement one or more style of communication between synchronous, asynchronous, and batch. This best practice helps you identify the different trade-offs associated with each style of communication to make your workload able to tolerate disruptions in any of its dependencies.

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

Implementation guidance

The following sections contain both general and specific implementation guidance for each kind of dependency.

General guidance

- Make sure that the performance and reliability service-level objectives (SLOs) that your dependencies offer meet the performance and reliability requirements of your workload.
- Use AWS observability services to monitor response times and error rates to make sure your dependency is providing service at the levels needed by your workload.
- Identify the potential challenges that your workload may face when communicating with its dependencies. Distributed systems come with a wide range of challenges that might increase architectural complexity, operational burden, and cost. Common challenges include latency, network disruptions, data loss, scaling, and data replication lag.
- Implement robust error handling and logging to help you troubleshoot problems when your dependency experiences issues.

Synchronous dependency

In synchronous communications, your workload sends a request to its dependency and blocks the operation waiting for a response. When its dependency receives the request, it tries to handle it as soon as possible and sends a response back to your workload. A significant challenge with synchronous communication is that it causes temporal coupling, which requires your workload and its dependencies to be available at the same time. When your workload needs to communicate synchronously with its dependencies, consider the following guidance:

- Your workload should not rely on multiple synchronous dependencies to perform a single function. This chain of dependencies increases overall brittleness because all dependencies in the pathway need to be available in order for the request to complete successfully.
- When a dependency is unhealthy or unavailable, determine your error handling and retry strategies. Avoid using bimodal behavior. Bimodal behavior is when your workload exhibits different behavior under normal and failure modes. For more details on bimodal behavior, see REL11-BP05 Use static stability to prevent bimodal behavior.
- Keep in mind that failing fast is better than making your workload wait. For instance, the AWS Lambda Developer Guide describes how to handle retries and failures when you invoke Lambda functions.
- Set timeouts when your workload calls its dependency. This technique avoids waiting too long or waiting indefinitely for a response. For helpful discussion of this issue, see Tuning AWS Java SDK HTTP request settings for latency-aware Amazon DynamoDB applications.
- Minimize the number of calls made from your workload to its dependency to fulfill a single request. Having chatty calls between them increases coupling and latency.

Asynchronous dependency

To temporally decouple your workload from its dependency, they should communicate asynchronously. Using an asynchronous approach, your workload can continue with any other processing without having to wait for its dependency, or chain of dependencies, to send a response.

When your workload needs to communicate asynchronously with its dependency, consider the following guidance:

- Determine whether to use messaging or event streaming based on your use case and requirements. Messaging allows your workload to communicate with its dependency by sending and receiving messages through a message broker. Event streaming allows your workload and its dependency to use a streaming service to publish and subscribe to events, delivered as continuous streams of data, that need to be processed as soon as possible.

- Messaging and event streaming handle messages differently so you need to make trade-off decisions based on:
  - **Message priority:** message brokers can process high-priority messages ahead of normal messages. In event streaming, all messages have the same priority.
  - **Message consumption**: message brokers ensure that consumers receive the message. Event streaming consumers must keep track of the last message they have read.
  - **Message ordering**: with messaging, receiving messages in the exact order they are sent is not guaranteed unless you use a first-in-first-out (FIFO) approach. Event streaming always preserves the order in which the data was produced.
  - **Message deletion**: with messaging, the consumer must delete the message after processing it. The event streaming service appends the message to a stream and remains in there until the message's retention period expires. This deletion policy makes event streaming suitable for replaying messages.
- Define how your workload knows when its dependency completes its work. For instance, when your workload invokes a Lambda function asynchronously, Lambda places the event in a queue and returns a success response without additional information. After processing is complete, the Lambda function can send the result to a destination, configurable based on success or failure.
- Build your workload to handle duplicate messages by leveraging idempotency. Idempotency means that the results of your workload do not change even if your workload is generated more than once for the same message. It is important to point out that messaging or streaming

services will redeliver a message if a network failure occurs or if an acknowledgement has not been received.

- If your workload does not get a response from its dependency, it needs to resubmit the request. Consider limiting the number of retries to preserve your workload's CPU, memory, and network resources to handle other requests. The AWS Lambda documentation shows how to handle errors for asynchronous invocation.
- Leverage suitable observability, debugging, and tracing tools to manage and operate your workload's asynchronous communication with its dependency. You can use Amazon CloudWatch to monitor messaging and event streaming services. You can also instrument your workload with AWS X-Ray to quickly gain insights for troubleshooting problems.

Batch dependency

Batch systems take input data, initiate a series of jobs to process it, and produce some output data, without manual intervention. Depending on the data size, jobs could run from minutes to, in some cases, several days. When your workload communicates with its batch dependency, consider the following guidance:

- Define the time window when your workload should run the batch job. Your workload can set up a recurrence pattern to invoke a batch system, for example, every hour or at the end of every month.
- Determine the location of the data input and the processed data output. Choose a storage service, such as Amazon Simple Storage Services (Amazon S3), Amazon Elastic File System (Amazon EFS), and Amazon FSx for Lustre, that allows your workload to read and write files at scale.
- If your workload needs to invoke multiple batch jobs, you could leverage AWS Step Functions to simplify the orchestration of batch jobs that run in AWS or on-premises. This sample project demonstrates orchestration of batch jobs using Step Functions, AWS Batch, and Lambda.
- Monitor batch jobs to look for abnormalities, such as a job taking longer than it should to complete. You could use tools like CloudWatch Container Insights to monitor AWS Batch environments and jobs. In this instance, your workload would stop the next job from beginning and inform the relevant staff of the exception.

Resources

Related documents:

- AWS Cloud Operations: Monitoring and Observability
- The Amazon's Builder Library: Challenges with distributed systems
- REL11-BP05 Use static stability to prevent bimodal behavior
- AWS Lambda Developer Guide: Error handling and automatic retries in AWS Lambda
- Tuning AWS Java SDK HTTP request settings for latency-aware Amazon DynamoDB applications
- AWS Messaging
- What is streaming data?
- AWS Lambda Developer Guide: Asynchronous invocation
- Amazon Simple Queue Service FAQ: FIFO queues
- Amazon Kinesis Data Streams Developer Guide: Handling Duplicate Records
- Amazon Simple Queue Service Developer Guide: Available CloudWatch metrics for Amazon SQS
- Amazon Kinesis Data Streams Developer Guide: Monitoring the Amazon Kinesis Data Streams Service with Amazon CloudWatch
- AWS X-Ray Developer Guide: AWS X-Ray concepts
- AWS Samples on GitHub: AWS Step functions Complex Orchestrator App
- AWS Batch User Guide: AWS Batch CloudWatch Container Insights

Related videos:

- AWS Summit SF 2022 - Full-stack observability and application monitoring with AWS (COP310)

Related tools:

- Amazon CloudWatch
- Amazon CloudWatch Logs
- AWS X-Ray
- Amazon Simple Storage Services (Amazon S3)
- Amazon Elastic File System (Amazon EFS)
- Amazon FSx for Lustre
- AWS Step Functions
- AWS Batch

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.

Decoupling dependencies, such as queuing systems, streaming systems, and workflows, help minimize the impact of changes or failure on a system. This separation isolates a component's behavior from affecting others that depend on it, improving resilience and agility.

In tightly coupled systems, changes to one component can necessitate changes in other components that rely on it, resulting in degraded performance across all components. *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 allows you to modify code or add features to a component while minimizing risk to other components that depend on it. It also allows for granular resilience at a component level where 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 Amazon SQS queue, a streaming data platform such as Amazon Kinesis, or AWS Step Functions.

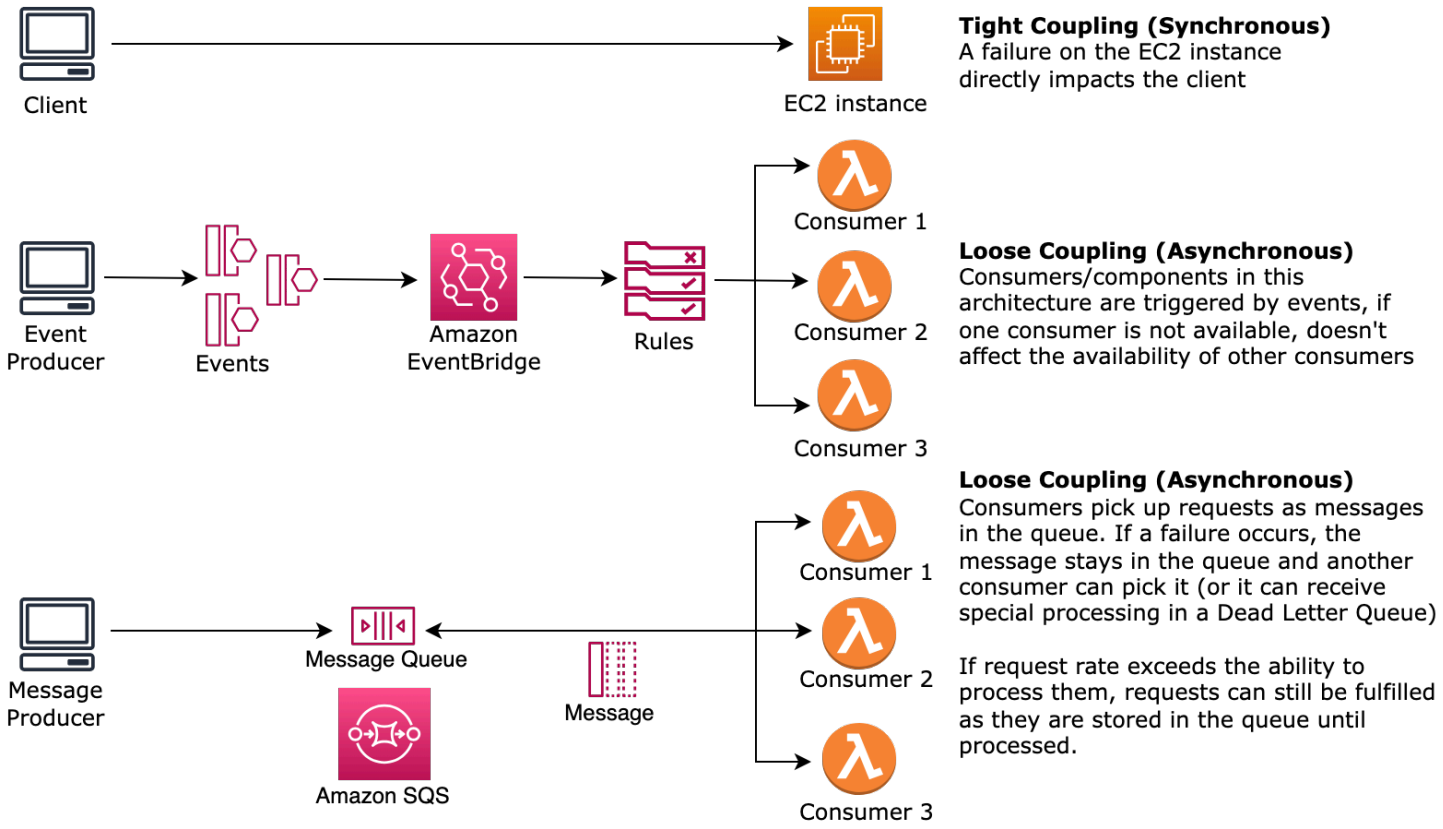


Figure 4: Dependencies such as queuing systems and load balancers are loosely coupled

Amazon SQS queues and AWS Step Functions 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.

Desired outcome: Implementing loosely coupled dependencies allows you to minimize the surface area for failure to a component level, which helps diagnose and resolve issues. It also simplifies development cycles, allowing teams to implement changes at a modular level without affecting the performance of other components that depend on it. This approach provides the capability

to scale out at a component level based on resource needs, as well as utilization of a component contributing to cost-effectiveness.

Common anti-patterns:

- Deploying a monolithic workload.
- Directly invoking APIs between workload tiers with no capability of failover or asynchronous processing of the request.
- Tight coupling using shared data. Loosely coupled systems should avoid sharing data through shared databases or other forms of tightly coupled data storage, which can reintroduce tight coupling and hinder scalability.
- Ignoring back pressure. Your workload should have the ability to slow down or stop incoming data when a component can't process it at the same rate.

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. There are various solutions that allow you to build loosely coupled applications. These include services for implementing fully managed queues, automated workflows, react to events, and APIs among others which can help isolate behavior of components from other components, and as such increasing resilience and agility.

- **Build event-driven architectures:** Amazon EventBridge helps you build loosely coupled and distributed event-driven architectures.
- **Implement queues in distributed systems:** You can use Amazon Simple Queue Service (Amazon SQS) to integrate and decouple distributed systems.
- **Containerize components as microservices:** Microservices allow teams to build applications composed of small independent components which communicate over well-defined APIs. Amazon Elastic Container Service (Amazon ECS), and Amazon Elastic Kubernetes Service (Amazon EKS) can help you get started faster with containers.
- **Manage workflows with Step Functions:** Step Functions help you coordinate multiple AWS services into flexible workflows.

- **Leverage publish-subscribe (pub/sub) messaging architectures:** Amazon Simple Notification Service (Amazon SNS) provides message delivery from publishers to subscribers (also known as producers and consumers).

Implementation steps

- Components in an event-driven architecture are initiated by events. Events are actions that happen in a system, such as a user adding an item to a cart. When an action is successful, an event is generated that actuates the next component of the system.
  - Building Event-driven Applications with Amazon EventBridge
  - AWS re:Invent 2022 - Designing Event-Driven Integrations using Amazon EventBridge
- Distributed messaging systems have three main parts that need to be implemented for a queue based architecture. They include components of the distributed system, the queue that is used for decoupling (distributed on Amazon SQS servers), and the messages in the queue. A typical system has producers which initiate the message into the queue, and the consumer which receives the message from the queue. The queue stores messages across multiple Amazon SQS servers for redundancy.
  - Basic Amazon SQS architecture
  - Send Messages Between Distributed Applications with Amazon Simple Queue Service
- Microservices, when well-utilized, enhance maintainability and boost scalability, as loosely coupled components are managed by independent teams. It also allows for the isolation of behaviors to a single component in case of changes.
  - Implementing Microservices on AWS
  - Let's Architect! Architecting microservices with containers
- With AWS Step Functions you can build distributed applications, automate processes, orchestrate microservices, among other things. The orchestration of multiple components into an automated workflow allows you to decouple dependencies in your application.
  - Create a Serverless Workflow with AWS Step Functions and AWS Lambda
  - Getting Started with AWS Step Functions

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?
- Break up with your monolith
- Orchestrate Queue-based Microservices with AWS Step Functions and Amazon SQS
- Basic Amazon SQS architecture
- Queue-Based Architecture

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
- AWS re:Invent 2022 - Designing event-driven integrations using Amazon EventBridge
- AWS re:Invent 2017: Elastic Load Balancing Deep Dive and Best Practices

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 mutating operations idempotent

An idempotent service promises that each request is processed exactly once, such that making multiple identical requests has the same effect as making a single request. This makes it easier for a client to implement retries without fear that a request is erroneously processed multiple times. To do this, clients can issue API requests with an idempotency token, which 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, even if the underlying state of the system has changed.

In a distributed system, it is relatively simple to perform an action at most once (client makes only one request) or at least once (keep requesting until client gets confirmation of success). It is more difficult to guarantee an action is 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 the need to create duplicate records or side effects.

Desired outcome: You have a consistent, well-documented, and widely adopted approach for ensuring idempotency across all components and services.

Common anti-patterns:

- You apply idempotency indiscriminately, even when not needed.
- You introduce overly complex logic for implementing idempotency.
- You use timestamps as keys for idempotency. This can cause inaccuracies due to clock skew or due to multiple clients that use the same timestamps to apply changes.
- You store entire payloads for idempotency. In this approach, you save complete data payloads for every request and overwrite it at each new request. This can degrade performance and affect scalability.
- You generate keys inconsistently across services. Without consistent keys, services may fail to recognize duplicate requests, which results in unintended results.

Benefits of establishing this best practice:

- Greater scalability: The system can handle retries and duplicate requests without having to perform additional logic or complex state management.

- Enhanced reliability: Idempotency helps services handle multiple identical requests in a consistent manner, which reduces the risk of unintended side effects or duplicate records. This is especially crucial in distributed systems, where network failures and retries are common.
- Improved data consistency: Because the same request produces the same response, idempotency helps maintain data consistency across distributed systems. This is essential to maintain the integrity of transactions and operations.
- Error handling: Idempotency tokens make error handling more straightforward. If a client does not receive a response due to an issue, it can safely resend the request with the same idempotency token.
- Operational transparency: Idempotency allows for better monitoring and logging. Services can log requests with their idempotency tokens, which makes it easier to trace and debug issues.
- Simplified API contract: It can simplify the contract between the client and server side systems and reduce the fear of erroneous data processing.

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

Implementation guidance

In a distributed system, performing an action at most once (the client makes only one request) or at least once (the client keeps requesting until success is confirmed) is relatively straightforward. However, it's challenging to implement *exactly once* behavior. To achieve this, your clients should generate and provide an idempotency token for each request.

By using idempotency tokens, a service can distinguish between new requests and repeated ones. When a service receives a request with an idempotency token, it checks if the token has already been used. If the token has been used, the service retrieves and returns the stored response. If the token is new, the service processes the request, stores the response along with the token, and then returns the response. This mechanism makes all responses idempotent, which enhances the reliability and consistency of the distributed system.

Idempotency is also an important behavior of event-driven architectures. These architectures are typically backed by a message queue such as Amazon SQS, Amazon MQ, Amazon Kinesis Streams, or Amazon Managed Streaming for Apache Kafka (MSK). In some circumstances, a message that was published only once may be accidentally delivered more than once. When a publisher generates and includes idempotency tokens in messages, it requests that the processing of any duplicate message received doesn't result in a repeated action for the same message. Consumers should keep track of each token received and ignore messages that contain duplicate tokens.

Services and consumers should also pass the received idempotency token to any downstream services that it calls. Every downstream service in the processing chain is similarly responsible for making sure that idempotency is implemented to avoid the side effect of processing a message more than once.

Implementation steps

1. **Identify idempotent operations**

   Determine which operations require idempotency. These typically include POST, PUT, and DELETE HTTP methods and database insert, update, or delete operations. Operations that do not mutate state, such as read-only queries, usually do not require idempotency unless they have side effects.

2. **Use unique identifiers**

   Include a unique token in each idempotent operation request sent by the sender, either directly in the request or as part of its metadata (for example, an HTTP header). This allows the receiver to recognize and handle duplicate requests or operations. Identifiers commonly used for tokens include Universally Unique Identifiers (UUIDs) and K-Sortable Unique Identifiers (KSUIDs).

3. **Track and manage state**

   Maintain the state of each operation or request in your workload. This can be achieved by storing the idempotency token and the corresponding state (such as pending, completed, or failed) in a database, cache, or other persistent store. This state information allows the workload to identify and handle duplicate requests or operations.

   Maintain consistency and atomicity by using appropriate concurrency control mechanisms if needed, such as locks, transactions, or optimistic concurrency controls. This includes the process of recording the idempotent token and running all mutating operations associated with servicing the request. This helps prevent race conditions and verifies that idempotent operations run correctly.

   Regularly remove old idempotency tokens from the datastore to manage storage and performance. If your storage system supports it, consider using expiration timestamps for data (often known as time to live, or TTL values). The likelihood of idempotency token reuse diminishes over time.

   Common AWS storage options typically used for storing idempotency tokens and related state include:

- **Amazon DynamoDB**: DynamoDB is a NoSQL database service that provides low-latency performance and high availability, which makes it well-suited for the storage of idempotency-related data. The key-value and document data model of DynamoDB allows for efficient storage and retrieval of idempotency tokens and associated state information. DynamoDB can also expire idempotency tokens automatically if your application sets a TTL value when it inserts them.
- **Amazon ElastiCache**: ElastiCache can store idempotency tokens with high throughput, low latency, and at low cost. Both ElastiCache (Redis) and ElastiCache (Memcached) can also expire idempotency tokens automatically if your application sets a TTL value when it inserts them.
- **Amazon Relational Database Service (RDS):** You can use Amazon RDS to store idempotency tokens and related state information, especially if your application already uses a relational database for other purposes.
- **Amazon Simple Storage Service (S3):** Amazon S3 is a highly scalable and durable object storage service that can be used to store idempotency tokens and related metadata. The versioning capabilities of S3 can be particularly useful for maintenance of the state of idempotent operations. The choice of storage service typically depends on factors such as the volume of idempotency-related data, the required performance characteristics, the need for durability and availability, and how the idempotency mechanism integrates with the overall workload architecture.

4. **Implement idempotent operations**

   Design your API and workload components to be idempotent. Incorporate idempotency checks into your workload components. Before you process a request or perform an operation, check if the unique identifier has already been processed. If it has, return the previous result instead of executing the operation again. For example, if a client sends a request to create a user, check if a user with the same unique identifier already exists. If the user exists, it should return the existing user information instead of creating a new one. Similarly, if a queue consumer receives a message with a duplicate idempotency token, the consumer should ignore the message.

   Create comprehensive test suites that validate the idempotency of requests. They should cover a wide range of scenarios, such as successful requests, failed requests, and duplicate requests.

   If your workload leverages AWS Lambda functions, consider Powertools for AWS Lambda. Powertools for AWS Lambda is a developer toolkit that helps implement serverless best practices and increase developer velocity when you work with AWS Lambda functions. In

particular, it provides a utility to convert your Lambda functions into idempotent operations which are safe to retry.

5. **Communicate idempotency clearly**

   Document your API and workload components to clearly communicate the idempotent nature of the operations. This helps clients understand the expected behavior and how to interact with your workload reliably.

6. **Monitor and audit**

   Implement monitoring and auditing mechanisms to detect any issues related to the idempotency of responses, such as unexpected response variations or excessive duplicate request handling. This can help you detect and investigate any issues or unexpected behaviors in your workload.

Resources

Related best practices:

- REL05-BP03 Control and limit retry calls
- REL06-BP01 Monitor all components for the workload (Generation)
- REL06-BP03 Send notifications (Real-time processing and alarming))
- REL08-BP02 Integrate functional testing as part of your deployment

Related documents:

- The Amazon Builders' Library: Making retries safe with idempotent APIs
- The Amazon Builders' Library: Challenges with distributed systems
- The Amazon Builders' Library: Reliability, constant work, and a good cup of coffee
- Amazon Elastic Container Service: Ensuring idempotency
- How do I make my Lambda function idempotent?
- Ensuring idempotency in Amazon EC2 API requests

Related videos:

- Building Distributed Applications with Event-driven Architecture - AWS Online Tech Talks

- AWS re:Invent 2023 - Building next-generation applications with event-driven architecture
- AWS re:Invent 2023 - Advanced integration patterns & trade-offs for loosely coupled systems
- AWS re:Invent 2023 - Advanced event-driven patterns with Amazon EventBridge
- 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)

Related tools:

- Idempotency with AWS Lambda Powertools (Java)
- Idempotency with AWS Lambda Powertools (Python)
- AWS Lambda Powertools GitHub page

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 allow 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
- REL05-BP02 Throttle requests
- REL05-BP03 Control and limit retry calls
- REL05-BP04 Fail fast and limit queues
- REL05-BP05 Set client timeouts
- REL05-BP06 Make systems stateless where possible
- REL05-BP07 Implement emergency levers

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

Application components should continue to perform their core function even if dependencies become unavailable. They might be serving slightly stale data, alternate data, or even no data. This ensures overall system function is only minimally impeded by localized failures while delivering the central business value.

Desired outcome: When a component's dependencies are unhealthy, the component itself can still function, although in a degraded manner. Failure modes of components should be seen as normal operation. Workflows should be designed in such a way that such failures do not lead to complete failure or at least to predictable and recoverable states.

Common anti-patterns:

- Not identifying the core business functionality needed. Not testing that components are functional even during dependency failures.
- Serving no data on errors or when only one out of multiple dependencies is unavailable and partial results can still be returned.
- Creating an inconsistent state when a transaction partially fails.
- Not having an alternative way to access a central parameter store.
- Invalidating or emptying local state as a result of a failed refresh without considering the consequences of doing so.

Benefits of establishing this best practice: Graceful degradation improves the availability of the system as a whole and maintains the functionality of the most important functions even during failures.

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

Implementation guidance

Implementing graceful degradation helps minimize the impact of dependency failures on component function. Ideally, a component detects dependency failures and works around them in a way that minimally impacts other components or customers.

Architecting for graceful degradation means considering potential failure modes during dependency design. For each failure mode, have a way to deliver most or at least the most

critical functionality of the component to callers or customers. These considerations can become additional requirements that can be tested and verified. Ideally, a component is able to perform its core function in an acceptable manner even when one or multiple dependencies fail.

This is as much a business discussion as a technical one. All business requirements are important and should be fulfilled if possible. However, it still makes sense to ask what should happen when not all of them can be fulfilled. A system can be designed to be available and consistent, but under circumstances where one requirement must be dropped, which one is more important? For payment processing, it might be consistency. For a real-time application, it might be availability. For a customer facing website, the answer may depend on customer expectations.

What this means depends on the requirements of the component and what should be considered its core function. For example:

- An ecommerce website might display data from multiple different systems like personalized recommendations, highest ranked products, and status of customer orders on the landing page. When one upstream system fails, it still makes sense to display everything else instead of showing an error page to a customer.
- A component performing batch writes can still continue processing a batch if one of the individual operations fails. It should be simple to implement a retry mechanism. This can be done by returning information on which operations succeeded, which failed, and why they failed to the caller, or putting failed requests into a dead letter queue to implement asynchronous retries. Information about failed operations should be logged as well.
- A system that processes transactions must verify that either all or no individual updates are executed. For distributed transactions, the saga pattern can be used to roll back previous operations in case a later operation of the same transaction fails. Here, the core function is maintaining consistency.
- Time critical systems should be able to deal with dependencies not responding in a timely manner. In these cases, the circuit breaker pattern can be used. When responses from a dependency start timing out, the system can switch to a closed state where no additional call are made.
- An application may read parameters from a parameter store. It can be useful to create container images with a default set of parameters and use these in case the parameter store is unavailable.

Note that the pathways taken in case of component failure need to be tested and should be significantly simpler than the primary pathway. Generally, fallback strategies should be avoided.

Implementation steps

Identify external and internal dependencies. Consider what kinds of failures can occur in them. Think about ways that minimize negative impact on upstream and downstream systems and customers during those failures.

The following is a list of dependencies and how to degrade gracefully when they fail:

1. **Partial failure of dependencies:** A component may make multiple requests to downstream systems, either as multiple requests to one system or one request to multiple systems each. Depending on the business context, different ways of handling for this may be appropriate (for more detail, see previous examples in Implementation guidance).
2. **A downstream system is unable to process requests due to high load:** If requests to a downstream system are consistently failing, it does not make sense to continue retrying. This may create additional load on an already overloaded system and make recovery more difficult. The circuit breaker pattern can be utilized here, which monitors failing calls to a downstream system. If a high number of calls are failing, it will stop sending more requests to the downstream system and only occasionally let calls through to test whether the downstream system is available again.
3. **A parameter store is unavailable:** To transform a parameter store, soft dependency caching or sane defaults included in container or machine images may be used. Note that these defaults need to be kept up-to-date and included in test suites.
4. **A monitoring service or other non-functional dependency is unavailable:** If a component is intermittently unable to send logs, metrics, or traces to a central monitoring service, it is often best to still execute business functions as usual. Silently not logging or pushing metrics for a long time is often not acceptable. Also, some use cases may require complete auditing entries to fulfill compliance requirements.
5. **A primary instance of a relational database may be unavailable:** Amazon Relational Database Service, like almost all relational databases, can only have one primary writer instance. This creates a single point of failure for write workloads and makes scaling more difficult. This can partially be mitigated by using a Multi-AZ configuration for high availability or Amazon Aurora Serverless for better scaling. For very high availability requirements, it can make sense to not rely on the primary writer at all. For queries that only read, read replicas can be used, which provide redundancy and the ability to scale out, not just up. Writes can be buffered, for example in an Amazon Simple Queue Service queue, so that write requests from customers can still be accepted even if the primary is temporarily unavailable.

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)

REL05-BP02 Throttle requests

Throttle requests to mitigate resource exhaustion due to unexpected increases in demand. Requests below throttling rates are processed while those over the defined limit are rejected with a return message indicating the request was throttled.

Desired outcome: Large volume spikes either from sudden customer traffic increases, flooding attacks, or retry storms are mitigated by request throttling, allowing workloads to continue normal processing of supported request volume.

Common anti-patterns:

- API endpoint throttles are not implemented or are left at default values without considering expected volumes.
- API endpoints are not load tested or throttling limits are not tested.
- Throttling request rates without considering request size or complexity.
- Testing maximum request rates or maximum request size, but not testing both together.

- Resources are not provisioned to the same limits established in testing.
- Usage plans have not been configured or considered for application to application (A2A) API consumers.
- Queue consumers that horizontally scale do not have maximum concurrency settings configured.
- Rate limiting on a per IP address basis has not been implemented.

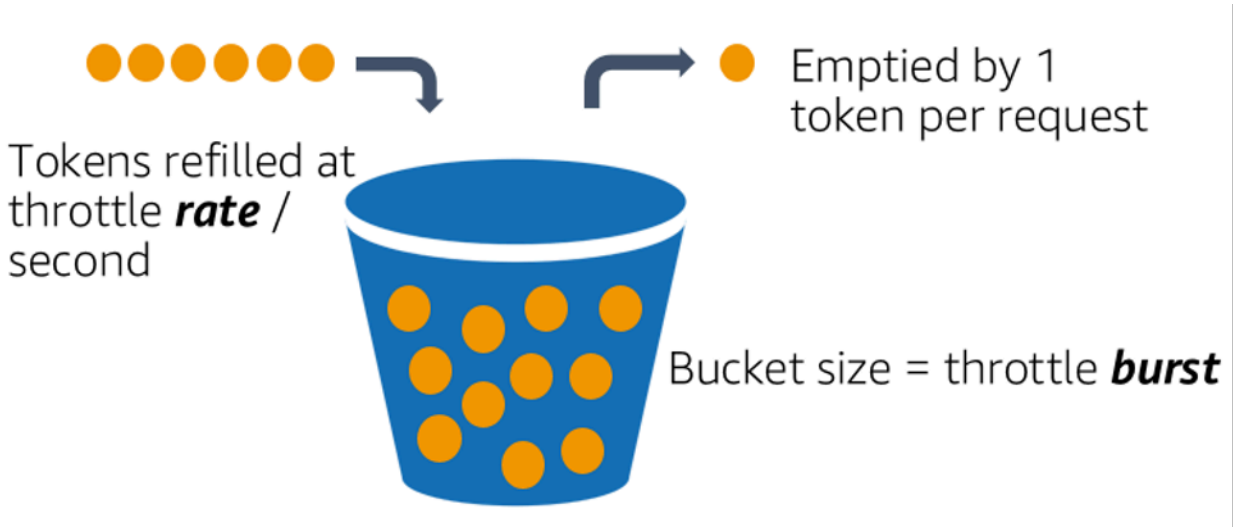
Benefits of establishing this best practice: Workloads that set throttle limits are able to operate normally and process accepted request load successfully under unexpected volume spikes. Sudden or sustained spikes of requests to APIs and queues are throttled and do not exhaust request processing resources. Rate limits throttle individual requestors so that high volumes of traffic from a single IP address or API consumer will not exhaust resources impact other consumers.

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

Implementation guidance

Services should be designed to process a known capacity of requests; this capacity can be established through load testing. If request arrival rates exceed limits, the appropriate response signals that a request has been throttled. This allows the consumer to handle the error and retry later.

When your service requires a throttling implementation, consider implementing the token bucket algorithm, where a token counts for a request. Tokens are refilled at a throttle rate per second and emptied asynchronously by one token per request.



The token bucket algorithm.

Amazon API Gateway implements the token bucket algorithm according to account and region limits and can be configured per-client with usage plans. Additionally, Amazon Simple Queue Service (Amazon SQS) and Amazon Kinesis can buffer requests to smooth out the request rate, and allow higher throttling rates for requests that can be addressed. Finally, you can implement rate limiting with AWS WAF to throttle specific API consumers that generate unusually high load.

Implementation steps

You can configure API Gateway with throttling limits for your APIs and return `429 Too Many Requests` errors when limits are exceeded. You can use AWS WAF with your AWS AppSync and API Gateway endpoints to enable rate limiting on a per IP address basis. Additionally, where your system can tolerate asynchronous processing, you can put messages into a queue or stream to speed up responses to service clients, which allows you to burst to higher throttle rates.

With asynchronous processing, when you've configured Amazon SQS as an event source for AWS Lambda, you can configure maximum concurrency to avoid high event rates from consuming available account concurrent execution quota needed for other services in your workload or account.

While API Gateway provides a managed implementation of the token bucket, in cases where you cannot use API Gateway, you can take advantage of language specific open-source implementations (see related examples in Resources) of the token bucket for your services.

- Understand and configure API Gateway throttling limits at the account level per region, API per stage, and API key per usage plan levels.
- Apply AWS WAF rate limiting rules to API Gateway and AWS AppSync endpoints to protect against floods and block malicious IPs. Rate limiting rules can also be configured on AWS AppSync API keys for A2A consumers.
- Consider whether you require more throttling control than rate limiting for AWS AppSync APIs, and if so, configure an API Gateway in front of your AWS AppSync endpoint.
- When Amazon SQS queues are set up as triggers for Lambda queue consumers, set maximum concurrency to a value that processes enough to meet your service level objectives but does not consume concurrency limits impacting other Lambda functions. Consider setting reserved concurrency on other Lambda functions in the same account and region when you consume queues with Lambda.
- Use API Gateway with native service integrations to Amazon SQS or Kinesis to buffer requests.

- If you cannot use API Gateway, look at language specific libraries to implement the token bucket algorithm for your workload. Check the examples section and do your own research to find a suitable library.
- Test limits that you plan to set, or that you plan to allow to be increased, and document the tested limits.
- Do not increase limits beyond what you establish in testing. When increasing a limit, verify that provisioned resources are already equivalent to or greater than those in test scenarios before applying the increase.

Resources

Related best practices:

- REL04-BP03 Do constant work
- REL05-BP03 Control and limit retry calls

Related documents:

- Amazon API Gateway: Throttle API Requests for Better Throughput
- AWS WAF: Rate-based rule statement
- Introducing maximum concurrency of AWS Lambda when using Amazon SQS as an event source
- AWS Lambda: Maximum Concurrency

Related examples:

- The three most important AWS WAF rate-based rules
- Java Bucket4j
- Python token-bucket
- Node token-bucket
- .NET System Threading Rate Limiting

Related videos:

- Implementing GraphQL API security best practices with AWS AppSync

Related tools:

- Amazon API Gateway
- AWS AppSync
- Amazon SQS
- Amazon Kinesis
- AWS WAF
- Virtual Waiting Room on AWS

REL05-BP03 Control and limit retry calls

Use exponential backoff to retry requests at progressively longer intervals between each retry. Introduce jitter between retries to randomize retry intervals. Limit the maximum number of retries.

Desired outcome: Typical components in a distributed software system include servers, load balancers, databases, and DNS servers. During normal operation, these components can respond to requests with errors that are temporary or limited, and also errors that would be persistent regardless of retries. When clients make requests to services, the requests consume resources including memory, threads, connections, ports, or any other limited resources. Controlling and limiting retries is a strategy to release and minimize consumption of resources so that system components under strain are not overwhelmed.

When client requests time out or receive error responses, they should determine whether or not to retry. If they do retry, they do so with exponential backoff with jitter and a maximum retry value. As a result, backend services and processes are given relief from load and time to self-heal, resulting in faster recovery and successful request servicing.

Common anti-patterns:

- Implementing retries without adding exponential backoff, jitter, and maximum retry values. Backoff and jitter help avoid artificial traffic spikes due to unintentionally coordinated retries at common intervals.
- Implementing retries without testing their effects or assuming retries are already built into an SDK without testing retry scenarios.
- Failing to understand published error codes from dependencies, leading to retrying all errors, including those with a clear cause that indicates lack of permission, configuration error, or another condition that predictably will not resolve without manual intervention.

- Not addressing observability practices, including monitoring and alerting on repeated service failures so that underlying issues are made known and can be addressed.
- Developing custom retry mechanisms when built-in or third-party retry capabilities suffice.
- Retrying at multiple layers of your application stack in a manner which compounds retry attempts further consuming resources in a retry storm. Be sure to understand how these errors affect your application the dependencies you rely on, then implement retries at only one level.
- Retrying service calls that are not idempotent, causing unexpected side effects like duplicated results.

Benefits of establishing this best practice: Retries help clients acquire desired results when requests fail but also consume more of a server's time to get the successful responses they want. When failures are rare or transient, retries work well. When failures are caused by resource overload, retries can make things worse. Adding exponential backoff with jitter to client retries allows servers to recover when failures are caused by resource overload. Jitter avoids alignment of requests into spikes, and backoff diminishes load escalation caused by adding retries to normal request load. Finally, it's important to configure a maximum number of retries or elapsed time to avoid creating backlogs that produce metastable failures.

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 retry intervals and limit the maximum number of retries.

Some AWS SDKs implement retries and exponential backoff by default. Use these built-in AWS implementations where applicable in your workload. Implement similar logic in your workload when calling services that are idempotent and where retries improve your client availability. Decide what the timeouts are and when to stop retrying based on your use case. Build and exercise testing scenarios for those retry use cases.

Implementation steps

- Determine the optimal layer in your application stack to implement retries for the services your application relies on.
- Be aware of existing SDKs that implement proven retry strategies with exponential backoff and jitter for your language of choice, and favor these over writing your own retry implementations.

- Verify that services are idempotent before implementing retries. Once retries are implemented, be sure they are both tested and regularly exercise in production.
- When calling AWS service APIs, use the AWS SDKs and AWS CLI and understand the retry configuration options. Determine if the defaults work for your use case, test, and adjust as needed.

Resources

Related best practices:

- REL04-BP04 Make mutating operations idempotent
- REL05-BP02 Throttle requests
- REL05-BP04 Fail fast and limit queues
- REL05-BP05 Set client timeouts
- REL11-BP01 Monitor all components of the workload to detect failures

Related documents:

- Error Retries and Exponential Backoff in AWS
- The Amazon Builders' Library: Timeouts, retries, and backoff with jitter
- Exponential Backoff and Jitter
- Making retries safe with idempotent APIs

Related examples:

- Spring Retry
- Resilience4j Retry

Related videos:

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

Related tools:

- AWS SDKs and Tools: Retry behavior
- AWS Command Line Interface: AWS CLI retries

REL05-BP04 Fail fast and limit queues

When a service is unable to respond successfully to a request, fail fast. This allows resources associated with a request to be released, and permits a service to recover if it's running out of resources. Failing fast is a well-established software design pattern that can be leveraged to build highly reliable workloads in the cloud. Queuing is also a well-established enterprise integration pattern that can smooth load and allow clients to release resources when asynchronous processing can be tolerated. When a service is able to respond successfully under normal conditions but fails when the rate of requests is too high, use a queue to buffer requests. However, do not allow a buildup of long queue backlogs that can result in processing stale requests that a client has already given up on.

Desired outcome: When systems experience resource contention, timeouts, exceptions, or grey failures that make service level objectives unachievable, fail fast strategies allow for faster system recovery. Systems that must absorb traffic spikes and can accommodate asynchronous processing can improve reliability by allowing clients to quickly release requests by using queues to buffer requests to backend services. When buffering requests to queues, queue management strategies are implemented to avoid insurmountable backlogs.

Common anti-patterns:

- Implementing message queues but not configuring dead letter queues (DLQ) or alarms on DLQ volumes to detect when a system is in failure.
- Not measuring the age of messages in a queue, a measurement of latency to understand when queue consumers are falling behind or erroring out causing retrying.
- Not clearing backlogged messages from a queue, when there is no value in processing these messages if the business need no longer exists.
- Configuring first in first out (FIFO) queues when last in first out (LIFO) queues would better serve client needs, for example when strict ordering is not required and backlog processing is delaying all new and time sensitive requests resulting in all clients experiencing breached service levels.
- Exposing internal queues to clients instead of exposing APIs that manage work intake and place requests into internal queues.

- Combining too many work request types into a single queue which can exacerbate backlog conditions by spreading resource demand across request types.
- Processing complex and simple requests in the same queue, despite needing different monitoring, timeouts and resource allocations.
- Not validating inputs or using assertions to implement fail fast mechanisms in software that bubble up exceptions to higher level components that can handle errors gracefully.
- Not removing faulty resources from request routing, especially when failures are grey emitting both successes and failures due to crashing and restarting, intermittent dependency failure, reduced capacity, or network packet loss.

Benefits of establishing this best practice: Systems that fail fast are easier to debug and fix, and often expose issues in coding and configuration before releases are published into production. Systems that incorporate effective queueing strategies provide greater resilience and reliability to traffic spikes and intermittent system fault conditions.

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

Implementation guidance

Fail fast strategies can be coded into software solutions as well as configured into infrastructure. In addition to failing fast, queues are a straightforward yet powerful architectural technique to decouple system components smooth load. Amazon CloudWatch provides capabilities to monitor for and alarm on failures. Once a system is known to be failing, mitigation strategies can be invoked, including failing away from impaired resources. When systems implement queues with Amazon SQS and other queue technologies to smooth load, they must consider how to manage queue backlogs, as well as message consumption failures.

Implementation steps

- Implement programmatic assertions or specific metrics in your software and use them to explicitly alert on system issues. Amazon CloudWatch helps you create metrics and alarms based on application log pattern and SDK instrumentation.
- Use CloudWatch metrics and alarms to fail away from impaired resources that are adding latency to processing or repeatedly failing to process requests.
- Use asynchronous processing by designing APIs to accept requests and append requests to internal queues using Amazon SQS and then respond to the message-producing client with a

success message so the client can release resources and move on with other work while backend queue consumers process requests.

- Measure and monitor for queue processing latency by producing a CloudWatch metric each time you take a message off a queue by comparing now to message timestamp.
- When failures prevent successful message processing or traffic spikes in volumes that cannot be processed within service level agreements, sideline older or excess traffic to a spillover queue. This allows priority processing of new work, and older work when capacity is available. This technique is an approximation of LIFO processing and allows normal system processing for all new work.
- Use dead letter or redrive queues to move messages that can't be processed out of the backlog into a location that can be researched and resolved later
- Either retry or, when tolerable, drop old messages by comparing now to the message timestamp and discarding messages that are no longer relevant to the requesting client.

Resources

Related best practices:

- REL04-BP02 Implement loosely coupled dependencies
- REL05-BP02 Throttle requests
- REL05-BP03 Control and limit retry calls
- REL06-BP02 Define and calculate metrics (Aggregation)
- REL06-BP07 Monitor end-to-end tracing of requests through your system

Related documents:

- Avoiding insurmountable queue backlogs
- Fail Fast
- How can I prevent an increasing backlog of messages in my Amazon SQS queue?
- Elastic Load Balancing: Zonal Shift
- Amazon Application Recovery Controller: Routing control for traffic failover

Related examples:

- Enterprise Integration Patterns: Dead Letter Channel

Related videos:

- AWS re:Invent 2022 - Operating highly available Multi-AZ applications

Related tools:

- Amazon SQS
- Amazon MQ
- AWS IoT Core
- Amazon CloudWatch

REL05-BP05 Set client timeouts

Set timeouts appropriately on connections and requests, verify them systematically, and do not rely on default values as they are not aware of workload specifics.

Desired outcome: Client timeouts should consider the cost to the client, server, and workload associated with waiting for requests that take abnormal amounts of time to complete. Since it is not possible to know the exact cause of any timeout, clients must use knowledge of services to develop expectations of probable causes and appropriate timeouts

Client connections time out based on configured values. After encountering a timeout, clients make decisions to back off and retry or open a circuit breaker. These patterns avoid issuing requests that may exacerbate an underlying error condition.

Common anti-patterns:

- Not being aware of system timeouts or default timeouts.
- Not being aware of normal request completion timing.
- Not being aware of possible causes for requests to take abnormally long to complete, or the costs to client, service, or workload performance associated with waiting on these completions.
- Not being aware of the probability of impaired network causing a request to fail only once timeout is reached, and the costs to client and workload performance for not adopting a shorter timeout.

- Not testing timeout scenarios both for connections and requests.
- Setting timeouts too high, which can result in long wait times and increase resource utilization.
- Setting timeouts too low, resulting in artificial failures.
- Overlooking patterns to deal with timeout errors for remote calls like circuit breakers and retries.
- Not considering monitoring for service call error rates, service level objectives for latency, and latency outliers. These metrics can provide insight to aggressive or permissive timeouts

Benefits of establishing this best practice: Remote call timeouts are configured and systems are designed to handle timeouts gracefully so that resources are conserved when remote calls respond abnormally slow and timeout errors are handled gracefully by service clients.

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 service dependency call and generally on any call across processes. Many frameworks offer built-in timeout capabilities, but be careful, as some have default values that are infinite or higher than acceptable for your service goals. 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 value that is too low 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.

Consider the following when determining timeout strategies:

- Requests may take longer than normal to process because of their content, impairments in a target service, or a networking partition failure.
- Requests with abnormally expensive content could consume unnecessary server and client resources. In this case, timing out these requests and not retrying can preserve resources. Services should also protect themselves from abnormally expensive content with throttles and server-side timeouts.
- Requests that take abnormally long due to a service impairment can be timed out and retried. Consideration should be given to service costs for the request and retry, but if the cause is a localized impairment, a retry is not likely to be expensive and will reduce client resource consumption. The timeout may also release server resources depending on the nature of the impairment.

- Requests that take a long time to complete because the request or response has failed to be delivered by the network can be timed out and retried. Because the request or response was not delivered, failure would have been the outcome regardless of the length of timeout. Timing out in this case will not release server resources, but it will release client resources and improve workload performance.

Take advantage of well-established design patterns like retries and circuit breakers to handle timeouts gracefully and support fail-fast approaches. AWS SDKs and AWS CLI allow for configuration of both connection and request timeouts and for retries with exponential backoff and jitter. AWS Lambda functions support configuration of timeouts, and with AWS Step Functions, you can build low code circuit breakers that take advantage of pre-built integrations with AWS services and SDKs. AWS App Mesh Envoy provides timeout and circuit breaker capabilities.

Implementation steps

- Configure timeouts on remote service calls and take advantage of built-in language timeout features or open source timeout libraries.
- When your workload makes calls with an AWS SDK, review the documentation for language specific timeout configuration.
  - Python
  - PHP
  - .NET
  - Ruby
  - Java
  - Go
  - Node.js
  - C++
- When using AWS SDKs or AWS CLI commands in your workload, configure default timeout values by setting the AWS configuration defaults for `connectTimeoutInMillis` and `tlsNegotiationTimeoutInMillis`.
- Apply command line options `cli-connect-timeout` and `cli-read-timeout` to control one-off AWS CLI commands to AWS services.
- Monitor remote service calls for timeouts, and set alarms on persistent errors so that you can proactively handle error scenarios.

- Implement CloudWatch Metrics and CloudWatch anomaly detection on call error rates, service level objectives for latency, and latency outliers to provide insight into managing overly aggressive or permissive timeouts.
- Configure timeouts on Lambda functions.
- API Gateway clients must implement their own retries when handling timeouts. API Gateway supports a 50 millisecond to 29 second integration timeout for downstream integrations and does not retry when integration requests timeout.
- Implement the circuit breaker pattern to avoid making remote calls when they are timing out. Open the circuit to avoid failing calls and close the circuit when calls are responding normally.
- For container based workloads, review App Mesh Envoy features to leverage built in timeouts and circuit breakers.
- Use AWS Step Functions to build low code circuit breakers for remote service calls, especially where calling AWS native SDKs and supported Step Functions integrations to simplify your workload.

Resources

Related best practices:

- REL05-BP03 Control and limit retry calls
- REL05-BP04 Fail fast and limit queues
- REL06-BP07 Monitor end-to-end tracing of requests through your system

Related documents:

- AWS SDK: Retries and Timeouts
- The Amazon Builders' Library: Timeouts, retries, and backoff with jitter
- Amazon API Gateway quotas and important notes
- AWS Command Line Interface: Command line options
- AWS SDK for Java 2.x: Configure API Timeouts
- AWS Botocore using the config object and Config Reference
- AWS SDK for .NET: Retries and Timeouts
- AWS Lambda: Configuring Lambda function options

Related examples:

- Using the circuit breaker pattern with AWS Step Functions and Amazon DynamoDB
- Martin Fowler: CircuitBreaker

Related tools:

- AWS SDKs
- AWS Lambda
- Amazon SQS
- AWS Step Functions
- AWS Command Line Interface

REL05-BP06 Make systems stateless where possible

Systems 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 allows servers to be replaced at will without causing an availability impact.

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.

Benefits of establishing this best practice: Systems that are designed to be stateless are more adaptable to horizontal scaling, making it possible to add or remove capacity based on fluctuating traffic and demand. They are also inherently resilient to failures and provide flexibility and agility in application development.

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

Implementation guidance

Make your applications stateless. Stateless applications allow horizontal scaling and are tolerant to the failure of an individual node. Analyze and understand the components of your application that maintain state within the architecture. This helps you assess the potential impact of transitioning to a stateless design. A stateless architecture decouples user data and offloads the session data. This provides the flexibility to scale each component independently to meet varying workload demands and optimize resource utilization.

Implementation steps

- Identify and understand the stateful components in your application.
- Decouple data by separating and managing user data from the core application logic.
  - Amazon Cognito can decouple user data from application code by using features, such as identity pools, user pools, and Amazon Cognito Sync.
  - You can use AWS Secrets Manager decouple user data by storing secrets in a secure, centralized location. This means that the application code doesn't need to store secrets, which makes it more secure.
  - Consider using Amazon S3 to store large, unstructured data, such as images and documents. Your application can retrieve this data when required, eliminating the need to store it in memory.
  - Use Amazon DynamoDB to store information such as user profiles. Your application can query this data in near-real time.
- Offload session data to a database, cache, or external files.
  - Amazon ElastiCache, Amazon DynamoDB, Amazon Elastic File System (Amazon EFS), and Amazon MemoryDB are examples of AWS services that you can use to offload session data.
- Design a stateless architecture after you identify which state and user data need to be persisted with your storage solution of choice.

Resources

Related best practices:

- REL11-BP03 Automate healing on all layers

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
- Best Practices for Stateless Web Tier on AWS

REL05-BP07 Implement emergency levers

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

Emergency levers work by disabling, throttling, or changing the behavior of components or dependencies using known and tested mechanisms. This can alleviate workload impairments caused by resource exhaustion due to unexpected increases in demand and reduce the impact of failures in non-critical components within your workload.

Desired outcome: By implementing emergency levers, you can establish known-good processes to maintain the availability of critical components in your workload. The workload should degrade gracefully and continue to perform its business-critical functions during the activation of an emergency lever. For more detail on graceful degradation, see REL05-BP01 Implement graceful degradation to transform applicable hard dependencies into soft dependencies.

Common anti-patterns:

- Failure of non-critical dependencies impacts the availability of your core workload.
- Not testing or verifying critical component behavior during non-critical component impairment.
- No clear and deterministic criteria defined for activation or deactivation of an emergency lever.

Benefits of establishing this best practice: Implementing emergency levers can improve the availability of the critical components in your workload by providing your resolvers with established processes to respond to unexpected spikes in demand or failures of non-critical dependencies.

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

Implementation guidance

- Identify critical components in your workload.
- Design and architect the critical components in your workload to withstand failure of non-critical components.

- Conduct testing to validate the behavior of your critical components during the failure of non-critical components.
- Define and monitor relevant metrics or triggers to initiate emergency lever procedures.
- Define the procedures (manual or automated) that comprise the emergency lever.

Implementation steps

- Identify business-critical components in your workload.
  - Each technical component in your workload should be mapped to its relevant business function and ranked as critical or non-critical. For examples of critical and non-critical functionality at Amazon, see Any Day Can Be Prime Day: How Amazon.com Search Uses Chaos Engineering to Handle Over 84K Requests Per Second.
  - This is both a technical and business decision, and varies by organization and workload.
- Design and architect the critical components in your workload to withstand failure of non-critical components.
  - During dependency analysis, consider all potential failure modes, and verify that your emergency lever mechanisms deliver the critical functionality to downstream components.
- Conduct testing to validate the behavior of your critical components during activation of your emergency levers.
  - Avoid bimodal behavior. For more detail, see REL11-BP05 Use static stability to prevent bimodal behavior.
- Define, monitor, and alert on relevant metrics to initiate the emergency lever procedure.
  - Finding the right metrics to monitor depends on your workload. Some example metrics are latency or the number of failed request to a dependency.
- Define the procedures, manual or automated, that comprise the emergency lever.
  - This may include mechanisms such as load shedding, throttling requests, or implementing graceful degradation.

Resources

Related best practices:

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

- REL05-BP02 Throttle requests
- REL11-BP05 Use static stability to prevent bimodal behavior

Related documents:

- Automating safe, hands-off deployments
- Any Day Can Be Prime Day: How Amazon.com Search Uses Chaos Engineering to Handle Over 84K Requests Per Second

Related videos:

- AWS re:Invent 2020: Reliability, consistency, and confidence through immutability

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 allows 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-BP01 Monitor all components for the workload (Generation)
- REL06-BP02 Define and calculate metrics (Aggregation)
- REL06-BP03 Send notifications (Real-time processing and alarming)
- REL06-BP04 Automate responses (Real-time processing and alarming)
- REL06-BP05 Analyze logs
- REL06-BP06 Regularly review monitoring scope and metrics
- REL06-BP07 Monitor end-to-end tracing of requests through your system

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 invoking 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 allows 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 allows 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. **Turn on logging where available.** Monitoring data should be obtained from all components of the workloads. Turn on additional logging, such as S3 Access Logs, and permit 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.
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 invoked.** 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

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:

- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility
- Observability workshop

REL06-BP02 Define and calculate metrics (Aggregation)

Collect metrics and logs from your workload components and calculate relevant aggregate metrics from them. These metrics provide broad and deep observability of your workload and can significantly improve your resilience posture.

Observability is more than just collecting metrics from workload components and being able to view and alert on them. It's about having a holistic understanding about your workload's behavior. This behavioral information comes from all components in your workloads, which includes the cloud services on which they depend, well-crafted logs, and metrics. This data gives you oversight on your workload's behavior as a whole, as well as an understanding of every component's interaction with every unit of work at a fine level of detail.

Desired outcome:

- You collect logs from your workload components and AWS service dependencies, and you publish them to a central location where they can be easily accessed and processed.
- Your logs contain high-fidelity and accurate timestamps.
- Your logs contain relevant information about the processing context, such as a trace identifier, user or account identifier, and remote IP address.
- You create aggregate metrics from your logs that represent your workload's behavior from a high-level perspective.
- You are able to query your aggregated logs to gain deep and relevant insights about your workload and identify actual and potential problems.

Common anti-patterns:

- You don't collect relevant logs or metrics from the compute instances your workloads run on or the cloud services they use.
- You overlook the collection of logs and metrics related to your business key performance indicators (KPIs).
- You analyze workload-related telemetry in isolation without aggregation and correlation.
- You allow metrics and logs to expire too quickly, which hinders trend analysis and recurring issue identification.

Benefits of establishing these best practices: You can detect more anomalies and correlate events and metrics between different components of your workload. You can create insights from your workload components based on information contained in logs that frequently aren't available in metrics alone. You can determine causes of failure more quickly by querying your logs at scale.

Level of risk exposed if these best practices are not established: High

Implementation guidance

Identify the sources of telemetry data that are relevant for your workloads and their components. This data comes not only from components that publish metrics, such as your operating system (OS) and application runtimes such as Java, but also from application and cloud service logs. For example, web servers typically log each request with detailed information such as the timestamp, processing latency, user ID, remote IP address, path, and query string. The level of detail in these logs helps you perform detailed queries and generate metrics that may not have been otherwise available.

Collect the metrics and logs using appropriate tools and processes. Logs generated by applications running on Amazon EC2 instance can be collected by an agent such as the Amazon CloudWatch Agent and published to a central storage service such as Amazon CloudWatch Logs. AWS-managed compute services such as AWS Lambda and Amazon Elastic Container Service publish logs to CloudWatch Logs for you automatically. Enable log collection for AWS storage and processing services used by your workloads such as Amazon CloudFront, Amazon S3, Elastic Load Balancing, and Amazon API Gateway.

Enrich your telemetry data with *dimensions* that can help you see behavioral patterns more clearly and isolate correlated problems to groups of related components. Once added, you can observe component behavior at a finer level of detail, detect correlated failures, and take appropriate remedial steps. Examples of useful dimensions include Availability Zone, EC2 instance ID, and container task or Pod ID.

Once you have collected the metrics and logs, you can write queries and generate aggregate metrics from them that provide useful insights into both normal and anomalous behavior. For example, you can use Amazon CloudWatch Logs Insights to derive custom metrics from your application logs, Amazon CloudWatch Metrics Insights to query your metrics at scale, Amazon CloudWatch Container Insights to collect, aggregate and summarize metrics and logs from your containerized applications and microservices, or Amazon CloudWatch Lambda Insights if you're using AWS Lambda functions. To create an aggregate error rate metric, you can increment a counter each time an error response or message is found in your component logs or calculate the aggregate value of an existing error rate metric. You can use this data to generate histograms that show *tail behavior*, such as the worst-performing requests or processes. You can also scan this data in real time for anomalous patterns using solutions such as CloudWatch Logs anomaly detection. These insights can be placed on dashboards to keep them organized according to your needs and preferences.

Querying logs can help you understand how specific requests were handled by your workload components and reveal request patterns or other context that has an impact on your workload's resilience. It can be useful to research and prepare queries in advance, based on your knowledge of how your applications and other components behave, so you can more easily run them as needed. For example, with CloudWatch Logs Insights, you can interactively search and analyze your log data stored in CloudWatch Logs. You can also use Amazon Athena to query logs from multiple sources, including many AWS services, at petabyte scale.

When you define a log retention policy, consider the value of historical logs. Historical logs can help identify long-term usage and behavioral patterns, regressions, and improvements in your workload's performance. Permanently deleted logs cannot be analyzed later. However, the value of historical logs tends to diminish over long periods of time. Choose a policy that balances your needs as appropriate and is compliant with any legal or contractual requirements you might be subject to.

Implementation steps

1. Choose collection, storage, analysis, and display mechanisms for your observability data.
2. Install and configure metric and log collectors on the appropriate components of your workload (for example, on Amazon EC2 instances and in sidecar containers). Configure these collectors to restart automatically if they unexpectedly stop. Enable disk or memory buffering for the collectors so that temporary publication failures don't impact your applications or result in lost data.
3. Enable logging on AWS services you use as a part of your workloads, and forward those logs to the storage service you selected if needed. Refer to the respective services' user or developer guides for detailed instructions.
4. Define the operational metrics relevant to your workloads that are based on your telemetry data. These could be based on direct metrics emitted from your workload components, which can include business KPI related metrics, or the results of aggregated calculations such as sums, rates, percentiles, or histograms. Calculate these metrics using your log analyzer, and place them on dashboards as appropriate.
5. Prepare appropriate log queries to analyze workload components, requests, or transaction behavior as needed.
6. Define and enable a log retention policy for your component logs. Periodically delete logs when they become older than the policy permits.

Resources

Related best practices:

- REL06-BP01 Monitor all components for the workload (Generation)
- REL06-BP03 Send notifications (Real-time processing and alarming)
- REL06-BP04 Automate responses (Real-time processing and alarming)
- REL06-BP05 Analyze logs
- REL06-BP06 Regularly review monitoring scope and metrics
- REL06-BP07 Monitor end-to-end tracing of requests through your system

Related documentation:

- How Amazon CloudWatch works
- Amazon Managed Prometheus
- Amazon Managed Grafana
- Analyzing log data with CloudWatch Logs Insights
- Amazon CloudWatch Lambda Insights
- Amazon CloudWatch Container Insights
- Query your metrics with CloudWatch Metrics Insights
- AWS Distro for OpenTelemetry
- Amazon CloudWatch Logs Insights Sample Queries
- Debugging with Amazon CloudWatch Synthetics and AWS X-Ray
- Searching and Filtering Log Data
- Sending Logs Directly to Amazon S3
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility

Related workshops:

- One Observability Workshop

Related tools:

- AWS Distro for OpenTelemetry (GitHub)

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

When organizations detect potential issues, they send real-time notifications and alerts to the appropriate personnel and systems in order to respond quickly and effectively to these issues.

Desired outcome: Rapid responses to operational events are possible through configuration of relevant alarms based on service and application metrics. When alarm thresholds are breached, the appropriate personnel and systems are notified so they can address underlying issues.

Common anti-patterns:

- Configuring alarms with an excessively high threshold, resulting in the failure to send vital notifications.
- Configuring alarms with a threshold that is too low, resulting in inaction on important alerts due to the noise of excessive notifications.
- Not updating alarms and their threshold when usage changes.
- For alarms best addressed through automated actions, sending the notification to personnel instead of generating the automated action results in excessive notifications being sent.

Benefits of establishing this best practice: Sending real-time notifications and alerts to the appropriate personnel and systems allows for early detection of issues and rapid responses to operational incidents.

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

Implementation guidance

Workloads should be equipped with real-time processing and alarming to improve the detectability of issues that could impact the availability of the application and serve as triggers for automated response. Organizations can perform real-time processing and alarming by creating alerts with defined metrics in order to receive notifications whenever significant events occur or a metric exceeds a threshold.

Amazon CloudWatch allows you to create metric and composite alarms using CloudWatch alarms based on static threshold, anomaly detection, and other criteria. For more detail on the types of alarms you can configure using CloudWatch, see the alarms section of the CloudWatch documentation.

You can construct customized views of metrics and alerts of your AWS resources for your teams using CloudWatch dashboards. The customizable home pages in the CloudWatch console allow you to monitor your resources in a single view across multiple Regions.

Alarms can perform one or more actions, like sending a notification to an Amazon SNS topic, performing an Amazon EC2 action or an Amazon EC2 Auto Scaling action, or creating an OpsItem or incident in AWS Systems Manager.

Amazon CloudWatch uses Amazon SNS to send notifications when the alarm changes state, providing message delivery from the publishers (producers) to the subscribers (consumers). For more detail on setting up Amazon SNS notifications, see Configuring Amazon SNS.

CloudWatch sends EventBridge events whenever a CloudWatch alarm is created, updated, deleted, or its state changes. You can use EventBridge with these events to create rules that perform actions, such as notifying you whenever the state of an alarm changes or automatically triggering events in your account using Systems Manager automation.

Stay informed with AWS Health. AWS Health is the authoritative source of information about the health of your AWS Cloud resources. Use AWS Health to get notified of any confirmed service events so you can quickly take steps to mitigate any impact. Create purpose-fit AWS Health event notifications to e-mail and chat channels through AWS User Notifications and integrate programmatically with your monitoring and alerting tools through Amazon EventBridge. If you use AWS Organizations, aggregate AWS Health events across accounts.

When should you use EventBridge or Amazon SNS?

Both EventBridge and Amazon SNS can be used to develop event-driven applications, and your choice will depend on your specific needs.

Amazon EventBridge is recommended when you want to build an application that reacts to events from your own applications, SaaS applications, and AWS services. EventBridge is the only event-based service that integrates directly with third-party SaaS partners. EventBridge also automatically ingests events from over 200 AWS services without requiring developers to create any resources in their account.

EventBridge uses a defined JSON-based structure for events, and helps you create rules that are applied across the entire event body to select events to forward to a target. EventBridge currently supports over 20 AWS services as targets, including AWS Lambda, Amazon SQS, Amazon SNS, Amazon Kinesis Data Streams, and Amazon Data Firehose.

Amazon SNS is recommended for applications that need high fan out (thousands or millions of endpoints). A common pattern we see is that customers use Amazon SNS as a target for their rule to filter the events that they need, and fan out to multiple endpoints.

Messages are unstructured and can be in any format. Amazon SNS supports forwarding messages to six different types of targets, including Lambda, Amazon SQS, HTTP/S endpoints, SMS, mobile push, and email. Amazon SNS typical latency is under 30 milliseconds. A wide range of AWS services send Amazon SNS messages by configuring the service to do so (more than 30, including Amazon EC2, Amazon S3, and Amazon RDS).

Implementation steps

1. Create an alarm using Amazon CloudWatch alarms.
   a. A metric alarm monitors a single CloudWatch metric or an expression dependent on CloudWatch metrics. The alarm initiates one or more actions based on the value of the metric or expression in comparison to a threshold over a number of time intervals. The action may consist of sending a notification to an Amazon SNS topic, performing an Amazon EC2 action or an Amazon EC2 Auto Scaling action, or creating an OpsItem or incident in AWS Systems Manager.
   b. A composite alarm consists of a rule expression that considers the alarm conditions of other alarms you've created. The composite alarm only enters alarm state if all rule conditions are met. The alarms specified in the rule expression of a composite alarm can include metric alarms and additional composite alarms. Composite alarms can send Amazon SNS notifications when their state changes and can create Systems Manager OpsItems or incidents when they enter the alarm state, but they cannot perform Amazon EC2 or Auto Scaling actions.
2. Set up Amazon SNS notifications. When creating a CloudWatch alarm, you can include an Amazon SNS topic to send a notification when the alarm changes state.
3. Create rules in EventBridge that matches specified CloudWatch alarms. Each rule supports multiple targets, including Lambda functions. For example, you can define an alarm that initiates when available disk space is running low, which triggers a Lambda function through an EventBridge rule, to clean up the space. For more detail on EventBridge targets, see EventBridge targets.

Resources

Related Well-Architected best practices:

- REL06-BP01 Monitor all components for the workload (Generation)
- REL06-BP02 Define and calculate metrics (Aggregation)
- REL12-BP01 Use playbooks to investigate failures

Related documents:

- Amazon CloudWatch
- CloudWatch Logs insights
- Using Amazon CloudWatch alarms
- Using Amazon CloudWatch dashboards
- Using Amazon CloudWatch metrics
- Setting up Amazon SNS notifications
- CloudWatch anomaly detection
- CloudWatch Logs data protection
- Amazon EventBridge
- Amazon Simple Notification Service

Related videos:

- reinvent 2022 observability videos
- AWS re:Invent 2022 - Observability best practices at Amazon

Related examples:

- One Observability Workshop
- Amazon EventBridge to AWS Lambda with feedback control by Amazon CloudWatch Alarms

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.

Automated real-time processing of alarms is implemented so that systems can take quick corrective action and attempt to prevent failures or degraded service when alarms are triggered.

Automated responses to alarms could include the replacement of failing components, the adjustment of compute capacity, the redirection of traffic to healthy hosts, availability zones, or other regions, and the notification of operators.

Desired outcome: Real-time alarms are identified, and automated processing of alarms is set up to invoke the appropriate actions taken to maintain service level objectives and service-level agreements (SLAs). Automation can range from self-healing activities of single components to full-site failover.

Common anti-patterns:

- Not having a clear inventory or catalog of key real-time alarms.
- No automated responses on critical alarms (for example, when compute is nearing exhaustion, autoscaling occurs).
- Contradictory alarm response actions.
- No standard operating procedures (SOPs) for operators to follow when they receive alert notifications.
- Not monitoring configuration changes, as undetected configuration changes can cause downtime for workloads.
- Not having a strategy to undo unintended configuration changes.

Benefits of establishing this best practice: Automating alarm processing can improve system resiliency. The system takes corrective actions automatically, reducing manual activities that allow for human, error-prone interventions. Workload operates meet availability goals, and reduces service disruption.

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

Implementation guidance

To effectively manage alerts and automate their response, categorize alerts based on their criticality and impact, document response procedures, and plan responses before ranking tasks.

Identify tasks requiring specific actions (often detailed in runbooks), and examine all runbooks and playbooks to determine which tasks can be automated. If actions can be defined, often they can be automated. If actions cannot be automated, document manual steps in an SOP and train operators on them. Continually challenge manual processes for automation opportunities where you can establish and maintain a plan to automate alert responses.

Implementation steps

1. **Create an inventory of alarms:** To obtain a list of all alarms, you can use the AWS CLI using the Amazon CloudWatch command `describe-alarms`. Depending upon how many alarms you have set up, you might have to use pagination to retrieve a subset of alarms for each call, or alternatively you can use the AWS SDK to obtain the alarms using an API call.
2. **Document all alarm actions:** Update a runbook with all alarms and their actions, irrespective if they are manual or automated. AWS Systems Manager provides predefined runbooks. For more information about runbooks, see Working with runbooks. For detail on how to view runbook content, see View runbook content.
3. **Set up and manage alarm actions:** For any of the alarms that require an action, specify the automated action using the CloudWatch SDK. For example, you can change the state of your Amazon EC2 instances automatically based on a CloudWatch alarm by creating and enabling actions on an alarm or disabling actions on an alarm.

   You can also use Amazon EventBridge to respond automatically to system events, such as application availability issues or resource changes. You can create rules to indicate which events you're interested in, and the actions to take when an event matches a rule. The actions that can be automatically initiated include invoking an AWS Lambda function, invoking Amazon EC2 Run Command, relaying the event to Amazon Kinesis Data Streams, and seeing Automate Amazon EC2 using EventBridge.
4. **Standard Operating Procedures (SOPs):** Based on your application components, AWS Resilience Hub recommends multiple SOP templates. You can use these SOPs to document all the processes an operator should follow in case an alert is raised. You can also construct a SOP based on Resilience Hub recommendations, where you need an Resilience Hub application with an associated resiliency policy, as well as a historic resiliency assessment against that application. The recommendations for your SOP are produced by the resiliency assessment.

   Resilience Hub works with Systems Manager to automate the steps of your SOPs by providing a number of SSM documents you can use as the basis for those SOPs. For example, Resilience Hub may recommend an SOP for adding disk space based on an existing SSM automation document.
5. **Perform automated actions using Amazon DevOps Guru:** You can use Amazon DevOps Guru to automatically monitor application resources for anomalous behavior and deliver targeted recommendations to speed up problem identification and remediation times. With DevOps Guru, you can monitor streams of operational data in near real time from multiple sources including Amazon CloudWatch metrics, AWS Config, AWS CloudFormation, and AWS X-Ray. You can

also use DevOps Guru to automatically create OpsItems in OpsCenter and send events to EventBridge for additional automation.

Resources

Related best practices:

- REL06-BP01 Monitor all components for the workload (Generation)
- REL06-BP02 Define and calculate metrics (Aggregation)
- REL06-BP03 Send notifications (Real-time processing and alarming)
- REL08-BP01 Use runbooks for standard activities such as deployment

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)

Related videos:

- AWS re:Invent 2022 - Observability best practices at Amazon
- AWS re:Invent 2020: Automate anything with AWS Systems Manager
- Introduction to AWS Resilience Hub
- Create Custom Ticket Systems for Amazon DevOps Guru Notifications
- Enable Multi-Account Insight Aggregation with Amazon DevOps Guru

Related examples:

- Amazon CloudWatch and Systems Manager Workshop

REL06-BP05 Analyze logs

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 allows 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 to 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 Regularly review monitoring scope and metrics

Frequently review how workload monitoring is implemented, and update it as your workload and its architecture evolves. Regular audits of your monitoring helps reduce the risk of missed or overlooked trouble indicators and further helps your workload meet its availability goals.

Effective monitoring is anchored in key business metrics, which evolve as your business priorities change. Your monitoring review process should emphasize service-level indicators (SLIs) and incorporate insights from your infrastructure, applications, clients, and users.

Desired outcome: You have an effective monitoring strategy that is regularly reviewed and updated periodically, as well as after any significant events or changes. You verify that key application health indicators are still relevant as your workload and business requirements evolve.

Common anti-patterns:

- You collect only default metrics.
- You set up a monitoring strategy, but you never review it.
- You don't discuss monitoring when major changes are deployed.

- You trust outdated metrics to determine workload health.
- Your operations teams are overwhelmed with false-positive alerts due to outdated metrics and thresholds.
- You lack observability of application components that are not being monitored.
- You focus only on low-level technical metrics and excluding business metrics in your monitoring.

Benefits of establishing this best practice: When you regularly review your monitoring, you can anticipate potential problems and verify that you are capable of detecting them. It also allows you to uncover blind spots that you might have missed during earlier reviews, which further improves your ability to detect issues.

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

Implementation guidance

Review monitoring metrics and scope during your operational readiness review (ORR) process. Perform periodic operational readiness reviews on a consistent schedule to evaluate whether there are any gaps between your current workload and the monitoring you have configured. Establish a regular cadence for operational performance reviews and knowledge sharing to enhance your ability to achieve higher performance from your operational teams. Validate whether existing alert thresholds are still adequate, and check for situations where operational teams are receiving false-positive alerts or not monitoring aspects of the application that should be monitored.

The Resilience Analysis Framework provides useful guidance that can help you navigate the process. The focus of the framework is to identify potential failure modes and the preventive and corrective controls you can use to mitigate their impact. This knowledge can help you identify the right metrics and events to monitor and alert upon.

Implementation steps

1. Schedule and conduct regular reviews of the workload dashboards. You may have different cadences for the depth at which you inspect.
2. 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 increased latency, decreased primary business function, and increased failure responses.
3. Inspect for outliers and anomalies in your metrics, which can be masked by averages or medians. Look at the highest and lowest values during the time frame, and investigate the causes of

observations that are far outside of normal bounds. As you continue to remove these causes, you can tighten your expected metric bounds in response to the improved consistency of your workload performance.

4. 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.
5. Review whether the current monitoring strategy remains relevant for the application. Based on an analysis of previous incidents (or the Resilience Analysis Framework), assess if there are additional aspects of the application that should be incorporated into the monitoring scope.
6. Review your Real User Monitoring (RUM) metrics to determine whether there are any gaps in application functionality coverage.
7. Review your change management process. Update your procedures if necessary to include a monitoring analysis step that should be performed before you approve a change.
8. Implement monitoring review as part of your operational readiness review and correction of error processes.

Resources

Related best practices

- REL06-BP01 Monitor all components for the workload (Generation)
- REL06-BP02 Define and calculate metrics (Aggregation)
- REL06-BP07 Monitor end-to-end tracing of requests through your system
- REL12-BP02 Perform post-incident analysis
- REL12-BP06 Conduct game days regularly

Related documents:

- Why you should develop a correction of error (COE)
- Using Amazon CloudWatch Dashboards
- Building dashboards for operational visibility
- Advanced Multi-AZ Resilience Patterns - Gray failures
- 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
- AWS Observability Best Practices
- Resilience Analysis Framework
- Resilience Analysis Framework - Observability
- Operational Readiness Review - ORR

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

Trace requests as they process through service components so product teams can more easily analyze and debug issues and improve performance.

Desired outcome: Workloads with comprehensive tracing across all components are easy to debug, improving mean time to resolution (MTTR) of errors and latency by simplifying root cause discovery. End-to-end tracing reduces the time it takes to discover impacted components and drill into the detailed root causes of errors or latency.

Common anti-patterns:

- Tracing is used for some components but not for all. For example, without tracing for AWS Lambda, teams might not clearly understand latency caused by cold starts in a spiky workload.
- Synthetic canaries or real-user monitoring (RUM) are not configured with tracing. Without canaries or RUM, client interaction telemetry is omitted from the trace analysis yielding an incomplete performance profile.
- Hybrid workloads include both cloud native and third party tracing tools, but steps have not been taken elect and fully integrate a single tracing solution. Based on the elected tracing solution, cloud native tracing SDKs should be used to instrument components that are not cloud native or third party tools should be configured to ingest cloud native trace telemetry.

Benefits of establishing this best practice: When development teams are alerted to issues, they can see a full picture of system component interactions, including component by component correlation to logging, performance, and failures. Because tracing makes it easy to visually identify root causes, less time is spent investigating root causes. Teams that understand component

interactions in detail make better and faster decisions when resolving issues. Decisions like when to invoke disaster recovery (DR) failover or where to best implement self-healing strategies can be improved by analyzing systems traces, ultimately improving customer satisfaction with your services.

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

Implementation guidance

Teams that operate distributed applications can use tracing tools to establish a correlation identifier, collect traces of requests, and build service maps of connected components. All application components should be included in request traces including service clients, middleware gateways and event buses, compute components, and storage, including key value stores and databases. Include synthetic canaries and real-user monitoring in your end-to-end tracing configuration to measure remote client interactions and latency so that you can accurately evaluate your systems performance against your service level agreements and objectives.

You can use AWS X-Ray and Amazon CloudWatch Application Monitoring instrumentation services to provide a complete view of requests as they travel through your application. X-Ray collects application telemetry and allows you to visualize and filter it across payloads, functions, traces, services, APIs, and can be turned on for system components with no-code or low-code. CloudWatch application monitoring includes ServiceLens to integrate your traces with metrics, logs, and alarms. CloudWatch application monitoring also includes synthetics to monitor your endpoints and APIs, as well as real-user monitoring to instrument your web application clients.

Implementation steps

- Use AWS X-Ray on all supported native services like Amazon S3, AWS Lambda, and Amazon API Gateway. These AWS services enable X-Ray with configuration toggles using infrastructure as code, AWS SDKs, or the AWS Management Console.
- Instrument applications AWS Distro for Open Telemetry and X-Ray or third-party collection agents.
- Review the AWS X-Ray Developer Guide for programming language specific implementation. These documentation sections detail how to instrument HTTP requests, SQL queries, and other processes specific to your application programming language.
- Use X-Ray tracing for Amazon CloudWatch Synthetic Canaries and Amazon CloudWatch RUM to analyze the request path from your end user client through your downstream AWS infrastructure.

- Configure CloudWatch metrics and alarms based on resource health and canary telemetry so that teams are alerted to issues quickly, and can then deep dive into traces and service maps with ServiceLens.
- Enable X-Ray integration for third party tracing tools like Datadog, New Relic, or Dynatrace if you are using third party tools for your primary tracing solution.

Resources

Related best practices:

- REL06-BP01 Monitor all components for the workload (Generation)
- REL11-BP01 Monitor all components of the workload to detect failures

Related documents:

- What is AWS X-Ray?
- Amazon CloudWatch: Application Monitoring
- Debugging with Amazon CloudWatch Synthetics and AWS X-Ray
- The Amazon Builders' Library: Instrumenting distributed systems for operational visibility
- Integrating AWS X-Ray with other AWS services
- AWS Distro for OpenTelemetry and AWS X-Ray
- Amazon CloudWatch: Using synthetic monitoring
- Amazon CloudWatch: Use CloudWatch RUM
- Set up Amazon CloudWatch synthetics canary and Amazon CloudWatch alarm
- Availability and Beyond: Understanding and Improving the Resilience of Distributed Systems on AWS

Related examples:

- One Observability Workshop

Related videos:

- AWS re:Invent 2022 - How to monitor applications across multiple accounts

- How to Monitor your AWS Applications

Related tools:

- AWS X-Ray
- Amazon CloudWatch
- Amazon Route 53

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
- REL07-BP02 Obtain resources upon detection of impairment to a workload
- REL07-BP03 Obtain resources upon detection that more resources are needed for a workload
- REL07-BP04 Load test your workload

REL07-BP01 Use automation when obtaining or scaling resources

A cornerstone of reliability in the cloud is the programmatic definition, provisioning, and management of your infrastructure and resources. Automation helps you streamline resource provisioning, facilitate consistent and secure deployments, and scale resources across your entire infrastructure.

Desired outcome: You manage your infrastructure as code (IaC). You define and maintain your infrastructure code in version control systems (VCS). You delegate provisioning AWS resources to automated mechanisms and leverage managed services like Application Load Balancer (ALB), Network Load Balancer (NLB), and Auto Scaling groups. You provision your resources using continuous integration/continuous delivery (CI/CD) pipelines so that code changes automatically initiate resource updates, including updates to your Auto Scaling configurations.

Common anti-patterns:

- You deploy resources manually using the command line or at the AWS Management Console (also known as *click-ops*).
- You tightly couple your application components or resources, and create inflexible architectures as a result.
- You implement inflexible scaling policies that do not adapt to changing business requirements, traffic patterns, or new resource types.
- You manually estimate capacity to meet anticipated demand.

Benefits of establishing this best practice: Infrastructure as code (IaC) allows infrastructure to be defined programmatically. This helps you manage infrastructure changes through the same software development lifecycle as application changes, which promotes consistency and repeatability and reduces the risk of manual, error-prone tasks. You can further streamline the process of provisioning and updating resources through implementing IaC with automated delivery pipelines. You can deploy infrastructure updates reliably and efficiently without the need for manual intervention. This agility is particularly important when scaling resources to meet fluctuating demands.

You can achieve dynamic, automated resource scaling in conjunction with IaC and delivery pipelines. By monitoring key metrics and applying predefined scaling policies, Auto Scaling can automatically provision or deprovision resources as needed, which improves performance and cost-efficiency. This reduces the potential for manual errors or delays in response to changes in application or workload requirements.

The combination of IaC, automated delivery pipelines, and Auto Scaling helps organizations provision, update, and scale their environments with confidence. This automation is essential to maintain a responsive, resilient, and efficiently-managed cloud infrastructure.

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

Implementation guidance

To set up automation with CI/CD pipelines and infrastructure as code (IaC) for your AWS architecture, choose a version control system such as Git to store your IaC templates and configuration. These templates can be written using tools such as AWS CloudFormation. To start, define your infrastructure components (such as AWS VPCs, Amazon EC2 Auto Scaling Groups, and Amazon RDS databases) within these templates.

Next, integrate these IaC templates with a CI/CD pipeline to automate the deployment process. AWS CodePipeline provides a seamless AWS-native solution, or you can use other third-party CI/CD

solutions. Create a pipeline that activates when changes occur to your version control repository. Configure the pipeline to include stages that lint and validate your IaC templates, deploy the infrastructure to a staging environment, run automated tests, and finally, deploy to production. Incorporate approval steps where necessary to maintain control over changes. This automated pipeline not only speeds up deployment but also facilitates consistency and reliability across environments.

Configure Auto Scaling of resources such as Amazon EC2 instances, Amazon ECS tasks, and database replicas in your IaC to provide automatic scale-out and scale-in as needed. This approach enhances application availability and performance and optimizes cost by dynamically adjusting resources based on demand. For a list of supported resources, see Amazon EC2 Auto Scaling and AWS Auto Scaling.

Implementation steps

1. Create and use a source code repository to store the code that controls your infrastructure configuration. Commit changes to this repository to reflect any ongoing changes you want to make.
2. Select an infrastructure as code solution such as AWS CloudFormation to keep your infrastructure up to date and detect inconsistency (drift) from your intended state.
3. Integrate your IaC platform with your CI/CD pipeline to automate deployments.
4. Determine and collect the appropriate metrics for automatic scaling of resources.
5. Configure automatic scaling of resources using scale-out and scale-in policies appropriate for your workload components. Consider using scheduled scaling for predictable usage patterns.
6. Monitor deployments to detect failures and regressions. Implement rollback mechanisms within your CI/CD platform to revert changes if necessary.

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
- 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?
- Integrating Jenkins with AWS CodeBuild and AWS CodeDeploy
- Creating a four stage pipeline with AWS CodePipeline

Related videos:

- Back to Basics: Deploy Your Code to Amazon EC2
- AWS Supports You | Starting Your Infrastructure as Code Solution Using AWS CloudFormation Templates
- Streamline Your Software Release Process Using AWS CodePipeline
- Monitor AWS Resources Using Amazon CloudWatch Dashboards
- Create Cross Account & Cross Region CloudWatch Dashboards | Amazon Web Services

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 start 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 through the AWS Management Console or AWS CLI). However, automation should be used whenever possible (refer to **Use automation when obtaining or scaling resources**).

Desired outcome: Scaling activities (either automatically or manually) are initiated to restore availability upon detection of a failure or degraded customer experience.

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

Implementation guidance

Implement observability and monitoring across all components in your workload, to monitor customer experience and detect failure. Define the procedures, manual or automated, that scale the required resources. o For more information, see REL11-BP01 Monitor all components of the workload to detect failures.

Implementation steps

- Define the procedures, manual or automated, that scale the required resources.
  - Scaling procedures depend on how the different components within your workload are designed.
  - Scaling procedures also vary depending on the underlying technology utilized.
    - Components using AWS Auto Scaling can use scaling plans 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. Auto Scaling provides recommendations for scaling strategies customized to each resource. After you create your scaling plan, Auto Scaling combines dynamic scaling and predictive scaling methods together to support your scaling strategy. For more detail, see How scaling plans work.
    - Amazon EC2 Auto Scaling verifies 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 and maximum number of instances in each Auto Scaling group, and Amazon EC2 Auto Scaling ensures that your group never goes below or above these limits. For more detail, see What is Amazon EC2 Auto Scaling?
    - Amazon DynamoDB auto scaling uses the Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your behalf, in response to actual traffic patterns. This allows a table or a global secondary index to increase its provisioned read and write capacity to handle sudden increases in traffic, without throttling. For more detail, see Managing throughput capacity automatically with DynamoDB auto scaling.

Resources

Related best practices:

- REL07-BP01 Use automation when obtaining or scaling resources
- REL11-BP01 Monitor all components of the workload to detect failures

Related documents:

- AWS Auto Scaling: How Scaling Plans Work
- 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

One of the most valuable features of cloud computing is the ability to provision resources dynamically.

In traditional on-premises compute environments, you must identify and provision enough capacity in advance to serve peak demand. This is a problem because it is expensive and because it poses risks to availability if you underestimate the workload's peak capacity needs.

In the cloud, you don't have to do this. Instead, you can provision compute, database, and other resource capacity as needed to meet current and forecasted demand. Automated solutions such as Amazon EC2 Auto Scaling and Application Auto Scaling can bring resources online for you based on metrics you specify. This can make the scaling process easier and predictable, and it can make your workload significantly more reliable by ensuring you have enough resources available at all times.

Desired outcome: You configure automatic scaling of compute and other resources to meet demand. You provide sufficient headroom in your scaling policies to allow bursts of traffic to be served while additional resources are brought online.

Common anti-patterns:

- You provision a fixed number of scalable resources.
- You choose a scaling metric that does not correlate to actual demand.
- You fail to provide enough headroom in your scaling plans to accommodate demand bursts.

- Your scaling policies add capacity too late, which leads to capacity exhaustion and degraded service while additional resources are brought online.
- You fail to correctly configure minimum and maximum resource counts, which leads to scaling failures.

Benefits of establishing this best practice: Having enough resources to meet current demand is critical to provide high availability of your workload and adhere to your defined service-level objectives (SLOs). Automatic scaling allows you to provide the right amount of compute, database, and other resources your workload needs in order to serve current and forecasted demand. You don't need to determine peak capacity needs and statically allocate resources to serve it. Instead, as demand grows, you can allocate more resources to accommodate it, and after demand falls, you can deactivate resources to reduce cost.

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

Implementation guidance

First, determine whether the workload component is suitable for automatic scaling. These components are called *horizontally scalable* because they provide the same resources and behave identically. Examples of horizontally-scalable components include EC2 instances that are configured alike, Amazon Elastic Container Service (ECS) tasks, and pods running on Amazon Elastic Kubernetes Service (EKS). These compute resources are typically located behind a load balancer and are referred to as *replicas*.

Other replicated resources may include database read replicas, Amazon DynamoDB tables, and Amazon ElastiCache (Redis OSS) clusters. For a complete list of supported resources, see AWS services that you can use with Application Auto Scaling.

For container-based architectures, you may need to scale two different ways. First, you may need to scale the containers that provide horizontally-scalable services. Second, you may need to scale the compute resources to make space for new containers. Different automatic scaling mechanisms exist for each layer. To scale ECS tasks, you can use Application Auto Scaling. To scale Kubernetes pods, you can use Horizontal Pod Autoscaler (HPA) or Kubernetes Event-driven Autoscaling (KEDA). To scale the compute resources, you can use Capacity Providers for ECS, or for Kubernetes, you can use Karpenter or Cluster Autoscaler.

Next, select how you will perform automatic scaling. There are three major options: metric-based scaling, scheduled scaling, and predictive scaling.

Metric-based scaling

Metric-based scaling provisions resources based on the value of one or more *scaling metrics*. A scaling metric is one that corresponds to your workload's demand. A good way to determine appropriate scaling metrics is to perform load testing in a non-production environment. During your load tests, keep the number of scalable resources fixed, and slowly increase demand (for example, throughput, concurrency, or simulated users). Then look for metrics that increase (or decrease) as demand grows, and conversely decrease (or increase) as demand falls. Typical scaling metrics include CPU utilization, work queue depth (such as an Amazon SQS queue), number of active users, and network throughput.

> **Note**
>
> AWS has observed that with most applications, memory utilization increases as the application warms up and then reaches a steady value. When demand decreases, memory utilization typically remains elevated rather than decreasing in parallel. Because memory utilization does not correspond to demand in both directions–that is, growing and falling with demand–consider carefully before you select this metric for automatic scaling.

Metric-based scaling is a *latent operation*. It can take several minutes for utilization metrics to propagate to auto scaling mechanisms, and these mechanisms typically wait for a clear signal of increased demand before reacting. Then, as the auto scaler creates new resources, it can take additional time for them to come to full service. Because of this, it is important to not set your scaling metric targets too close to full utilization (for example, 90% CPU utilization). Doing so risks exhausting existing resource capacity before additional capacity can come online. Typical resource utilization targets can range between 50-70% for optimum availability, depending on demand patterns and time required to provision additional resources.

Scheduled scaling

Scheduled scaling provisions or removes resources based on the calendar or time of day. It is frequently used for workloads that have predictable demand, such as peak utilization during weekday business hours or sales events. Both Amazon EC2 Auto Scaling and Application Auto Scaling support scheduled scaling. KEDA's cron scaler supports scheduled scaling of Kubernetes pods.

Predictive scaling

Predictive scaling uses machine learning to automatically scale resources based on anticipated demand. Predictive scaling analyzes the historical value of a utilization metric you provide and continuously predicts its future value. The predicted value is then used to scale the resource up or down. Amazon EC2 Auto Scaling can perform predictive scaling.

Implementation steps

1. Determine whether the workload component is suitable for automatic scaling.
2. Determine what kind of scaling mechanism is most appropriate for the workload: metric-based scaling, scheduled scaling, or predictive scaling.
3. Select the appropriate automatic scaling mechanism for the component. For Amazon EC2 instances, use Amazon EC2 Auto Scaling. For other AWS services, use Application Auto Scaling. For Kubernetes pods (such as those running in an Amazon EKS cluster), consider Horizontal Pod Autoscaler (HPA) or Kubernetes Event-driven Autoscaling (KEDA). For Kubernetes or EKS nodes, consider Karpenter and Cluster Auto Scaler (CAS).
4. For metric or scheduled scaling, conduct load testing to determine the appropriate scaling metrics and target values for your workload. For scheduled scaling, determine the number of resources needed at the dates and times you select. Determine the maximum number of resources needed to serve expected peak traffic.
5. Configure the auto scaler based on the information collected above. Consult the auto scaling service's documentation for details. Verify that the maximum and minimum scaling limits are configured correctly.
6. Verify the scaling configuration is working as expected. Perform load testing in a non-production environment and observe how the system reacts, and adjust as needed. When enabling auto scaling in production, configure appropriate alarms to notify you of any unexpected behavior.

Resources

Related documents:

- What Is Amazon EC2 Auto Scaling?
- AWS Prescriptive Guidance: Load testing applications
- 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

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
- Load testing applications

Related videos:

- AWS Summit ANZ 2023: Accelerate with confidence through AWS Distributed Load Testing

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.

Additional deployment patterns to minimize risk

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

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

Operational Readiness Reviews (ORRs)

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

Best practices

- REL08-BP01 Use runbooks for standard activities such as deployment
- REL08-BP02 Integrate functional testing as part of your deployment
- REL08-BP03 Integrate resiliency testing as part of your deployment
- REL08-BP04 Deploy using immutable infrastructure
- REL08-BP05 Deploy changes with automation

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 invoke 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 run 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

- Provide 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 a hosted source code management system based on a popular technology such as Git to store your source code and infrastructure as code (IaC) configuration.

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?

Related examples:

- Automating operations with Playbooks and Runbooks

REL08-BP02 Integrate functional testing as part of your deployment

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

Unit testing is the process where you test the smallest functional unit of code to validate its behavior. Integration testing seeks to validate that each application feature works according to the software requirements. While unit tests focus on testing part of an application in isolation, integration tests consider side effects (for example, the effect of data being changed through a mutation operation). In either case, tests should be integrated into a deployment pipeline, and 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.

You achieve the best outcomes when these tests are run automatically as part of build and deployment actions. For instance, with AWS CodePipeline, developers commit changes to a source repository where CodePipeline automatically detects the changes. The application is built, and unit tests are run. After the unit tests have passed, 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.

Desired outcome: You use automation to perform unit and integration tests to validate that your code behaves as expected. These tests are integrated into the deployment process, and a test failure aborts the deployment.

Common anti-patterns:

- You ignore or bypass test failures and plans during the deployment process in order to accelerate the deployment timeline.

- You manually perform tests outside the deployment pipeline.
- You skip testing steps in the automation through manual emergency workflows.
- You run automated tests in an environment that does not closely resemble the production environment.
- You build a test suite that is insufficiently flexible and is difficult to maintain, update, or scale as the application evolves.

Benefits of establishing this best practice: Automated testing during the deployment process catches issues early, which reduces the risk of a release to production with bugs or unexpected behavior. Unit tests validate the code behaves as desired and API contracts are honored. Integration tests validate that the system operates according to specified requirements. These types of tests verify the intended working order of components such as user interfaces, APIs, databases, and source code.

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

Implementation guidance

Adopt a test-driven development (TDD) approach to writing software, where you develop test cases to specify and validate your code. To start, create test cases for each function. If the test fails, you write new code to pass the test. This approach helps you validate the expected result of each function. Run unit tests and validate that they pass before you commit code to a source code repository.

Implement both unit and integration tests as part of the build, test, and deployment stages of the CI/CD pipeline. Automate testing, and automatically initiate tests whenever a new version of the application is ready to be deployed. If success criteria are not met, the pipeline is halted or rolled back.

If the application is a web or mobile app, perform automated integration testing on multiple desktop browsers or real devices. This approach is particularly useful to validate the compatibility and functionality of mobile apps across a diverse range of devices.

Implementation steps

1. Write unit tests before you write functional code (*test-driven development*, or TDD). Establish code guidelines so that writing and running unit tests are a non-functional coding requirement.

2. Create a suite of automated integration tests that cover the identified testable functionalities. These tests should simulate user interactions and validate the expected outcomes.
3. Create the necessary test environment to run the integration tests. This may include staging or pre-production environments that closely mimic the production environment.
4. Set up your source, build, test, and deploy stages using the AWS CodePipeline console or AWS Command Line Interface (CLI).
5. Deploy the application once the code has been built and tested. AWS CodeDeploy can deploy it to your staging (testing) and production environments. These environments may include Amazon EC2 instances, AWS Lambda functions, or on-premises servers. The same deployment mechanism should be used to deploy the application to all environments.
6. Monitor the progress of your pipeline and the status of each stage. Use quality checks to block the pipeline based on the status of your tests. You can also receive notifications for any pipeline stage failure or pipeline completion.
7. Continually monitor the results of the tests, and look for patterns, regressions or areas that require more attention. Use this information to improve the test suite, identify areas of the application that need more robust testing, and optimize the deployment process.

Resources

Related best practices:

- REL07-BP04 Load test your workload
- REL08-BP03 Integrate resiliency testing as part of your deployment
- REL12-BP04 Test resiliency using chaos engineering

Related documents:

- AWS Prescriptive Guidance: Test automation
- Continuous Delivery and Continuous Integration
- Indicators for functional testing
- Monitoring pipelines
- Use AWS CodePipeline with AWS CodeBuild to test code and run builds
- AWS Device Farm

REL08-BP03 Integrate resiliency testing as part of your deployment

Integrate resiliency testing by consciously introducing failures in your system to measure its capability in case of disruptive scenarios. Resilience tests are different from unit and function tests that are usually integrated in deployment cycles, as they focus on the identification of unanticipated failures in your system. While it is safe to start with resiliency testing integration in pre-production, set a goal to implement these tests in production as a part of your *game days*.

Desired outcome: Resiliency testing helps build confidence in the system's ability to withstand degradation in production. Experiments identify weak points that could lead to failure, which helps you improve your system to automatically and efficiently mitigate failure and degradation.

Common anti-patterns:

- Lack of observability and monitoring in deployment processes
- Reliance on humans to resolve system failures
- Poor quality analysis mechanisms
- Focus on known issues in a system and a lack of experimentation to identify any unknowns
- Identification of failures, but no resolution
- No documentation of findings and runbooks

Benefits of establishing best practices: Resilience testing integrated in your deployments helps to identify unknown issues in the system that otherwise go unnoticed, which can lead to downtime in production. Identification of these unknowns in a system helps you document findings, integrate testing into your CI/CD process, and build runbooks, which simplify mitigation through efficient, repeatable mechanisms.

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

Implementation guidance

The most common resiliency testing forms that can be integrated in your system's deployments are disaster recovery and chaos engineering.

- Include updates to your disaster recovery plans and standard operating procedures (SOPs) with any significant deployment.

- Integrate reliability testing into your automated deployment pipelines. Services such asAWS Resilience Hubcan be integrated into your CI/CD pipeline to establish continuous resilience assessments that are automatically evaluated as part of every deployment.
- Define your applications in AWS Resilience Hub. Resilience assessments generate code snippets that help you create recovery procedures as AWS Systems Manager documents for your applications and provide a list of recommended Amazon CloudWatch monitors and alarms.
- Once your DR plans and SOPs are updated, complete disaster recovery testing to verify that they are effective. Disaster recovery testing helps you determine if you can restore your system after an event and return to normal operations. You can simulate various disaster recovery strategies and identify whether your planning is sufficient to meet your uptime requirements. Common disaster recovery strategies include backup and restore, pilot light, cold standby, warm standby, hot standby, and active-active, and they all differ in cost and complexity. Before disaster recovery testing, we recommend that you define your recovery time objective (RTO) and recovery point objective (RPO) to simplify the choice of strategy to simulate. AWS offers disaster recovery tools like AWS Elastic Disaster Recovery to help you get started with your planning and testing.
- Chaos engineering experiments introduce disruptions to the system, such as network outages and service failures. By simulating with controlled failures, you can discover your system's vulnerabilities while containing the impacts of the injected failures. Just like the other strategies, run controlled failure simulations in non-production environments using services like AWS Fault Injection Service to gain confidence before deploying in production.

Resources

Related documents:

- Experiment with failure using resilience testing to build recovery preparedness
- Continually assessing application resilience with AWS Resilience Hub and AWS CodePipeline
- Disaster recovery (DR) architecture on AWS, part 1: Strategies for recovery in the cloud
- Verify the resilience of your workloads using Chaos Engineering
- Principles of Chaos Engineering
- Chaos Engineering Workshop

Related videos:

- AWS re:Invent 2020: Testing Resilience using Chaos Engineering

- Improve Application Resilience with AWS Fault Injection Service
- Prepare & Protect Your Applications From Disruption With AWS Resilience Hub

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.

Follow an immutable infrastructure deployment strategy to increase the reliability, consistency, and reproducibility in your workload deployments.

Desired outcome: With immutable infrastructure, no in-place modifications are allowed to run infrastructure resources within a workload. Instead, when a change is needed, a new set of updated infrastructure resources containing all the necessary changes are deployed in parallel to your existing resources. This deployment is validated automatically, and if successful, traffic is gradually shifted to the new set of resources.

This deployment strategy applies to software updates, security patches, infrastructure changes, configuration updates, and application updates, among others.

Common anti-patterns:

- Implementing in-place changes to running infrastructure resources.

Benefits of establishing this best practice:

- **Increased consistency across environments:** Since there are no differences in infrastructure resources across environments, consistency is increased and testing is simplified.
- **Reduction in configuration drifts:** By replacing infrastructure resources with a known and version-controlled configuration, the infrastructure is set to a known, tested, and trusted state, avoiding configuration drifts.
- **Reliable atomic deployments:** Deployments either complete successfully or nothing changes, increasing consistency and reliability in the deployment process.
- **Simplified deployments:** Deployments are simplified because they don't need to support upgrades. Upgrades are just new deployments.

- **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.
- **Enhanced security posture:** By not allowing changes to infrastructure, remote access mechanisms (such as SSH) can be disabled. 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

Automation

When defining an immutable infrastructure deployment strategy, it is recommended to use automation as much as possible to increase reproducibility and minimize the potential of human error. For more detail, see REL08-BP05 Deploy changes with automation and Automating safe, hands-off deployments.

With infrastructure as code (IaC), infrastructure provisioning, orchestration, and deployment steps are defined in a programmatic, descriptive, and declarative way and stored in a source control system. Leveraging infrastructure as code makes it simpler to automate infrastructure deployment and helps achieve infrastructure immutability.

Deployment patterns

When a change in the workload is required, the immutable infrastructure deployment strategy mandates that a new set of infrastructure resources is deployed, including all necessary changes. It is important for this new set of resources to follow a rollout pattern that minimizes user impact. There are two main strategies for this deployment:

Canary deployment: 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 allows a canary deployment.

Blue/green deployment: 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 allows a blue/green deployment.

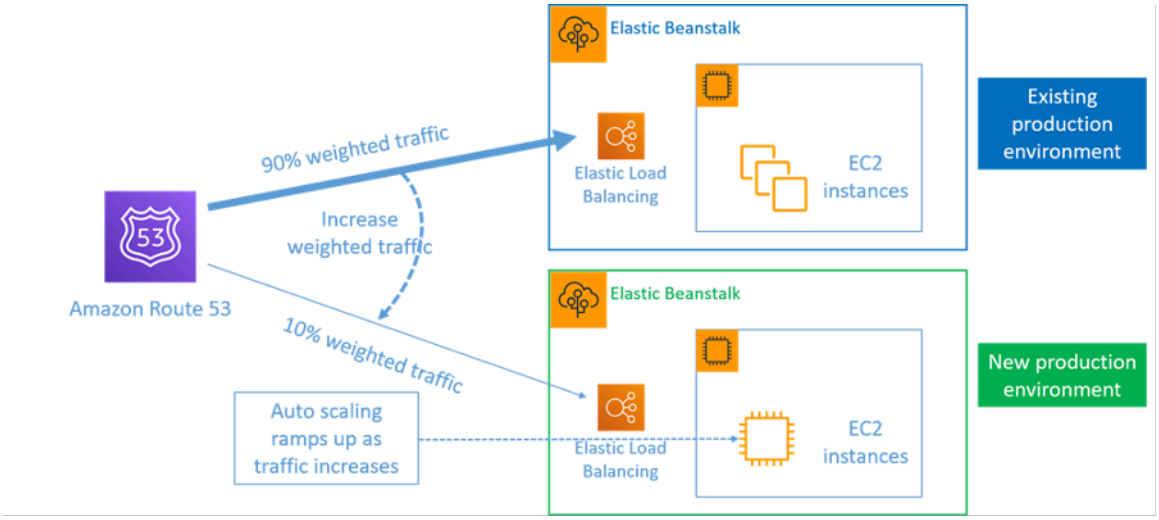


Figure 8: Blue/green deployment with AWS Elastic Beanstalk and Amazon Route 53

Drift detection

Drift is defined as any change that causes an infrastructure resource to have a different state or configuration to what is expected. Any type of unmanaged configuration change goes against the notion of immutable infrastructure, and should be detected and remediated in order to have a successful implementation of immutable infrastructure.

Implementation steps

- Disallow the in-place modification of running infrastructure resources.
  - You can use AWS Identity and Access Management (IAM) to specify who or what can access services and resources in AWS, centrally manage fine-grained permissions, and analyze access to refine permissions across AWS.
- Automate the deployment of infrastructure resources to increase reproducibility and minimize the potential of human error.
  - As described in the Introduction to DevOps on AWS whitepaper, automation is a cornerstone with AWS services and is internally supported in all services, features, and offerings.
  - *Prebaking* your Amazon Machine Image (AMI) can speed up the time to launch them. EC2 Image Builder is a fully managed AWS service that helps you automate the creation,

maintenance, validation, sharing, and deployment of customized, secure, and up-to-date Linux or Windows custom AMI.

- Some of the services that support automation are:
  - AWS Elastic Beanstalk is a service to rapidly deploy and scale web applications developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, NGINX, Passenger, and IIS.
  - AWS Proton helps platform teams connect and coordinate all the different tools your development teams need for infrastructure provisioning, code deployments, monitoring, and updates. AWS Proton enables automated infrastructure as code provisioning and deployment of serverless and container-based applications.
- Leveraging infrastructure as code makes it easy to automate infrastructure deployment, and helps achieve infrastructure immutability. AWS provides services that enable the creation, deployment, and maintenance of infrastructure in a programmatic, descriptive, and declarative way.
  - AWS CloudFormation helps developers create AWS resources in an orderly and predictable fashion. Resources are written in text files using JSON or YAML format. The templates require a specific syntax and structure that depends on the types of resources being created and managed. You author your resources in JSON or YAML with any code editor, check it into a version control system, and then AWS CloudFormation builds the specified services in safe, repeatable manner.
  - AWS Serverless Application Model (AWS SAM) is an open-source framework that you can use to build serverless applications on AWS. AWS SAM integrates with other AWS services, and is an extension of AWS CloudFormation.
  - AWS Cloud Development Kit (AWS CDK) is an open-source software development framework to model and provision your cloud application resources using familiar programming languages. You can use AWS CDK to model application infrastructure using TypeScript, Python, Java, and .NET. AWS CDK uses AWS CloudFormation in the background to provision resources in a safe, repeatable manner.
  - AWS Cloud Control API introduces a common set of Create, Read, Update, Delete, and List (CRUDL) APIs to help developers manage their cloud infrastructure in an easy and consistent way. The Cloud Control API common APIs allow developers to uniformly manage the lifecycle of AWS and third-party services.
- Implement deployment patterns that minimize user impact.
  - Canary deployments:

    - Set up an API Gateway canary release deployment
    - Create a pipeline with canary deployments for Amazon ECS using AWS App Mesh
  - Blue/green deployments: the Blue/Green Deployments on AWS whitepaper describes example techniques to implement blue/green deployment strategies.
- Detect configuration or state drifts. For more detail, see Detecting unmanaged configuration changes to stacks and resources.

Resources

Related best practices:

- REL08-BP05 Deploy changes with automation

Related documents:

- Automating safe, hands-off deployments
- Leveraging AWS CloudFormation to create an immutable infrastructure at Nubank
- Infrastructure as code
- Implementing an alarm to automatically detect drift in AWS CloudFormation stacks

Related videos:

- AWS re:Invent 2020: Reliability, consistency, and confidence through immutability

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.

Desired outcome: You build automated deployment safety into the release process with extensive pre-production testing, automatic rollbacks, and staggered production deployments. This

automation minimizes the potential impact on production caused by failed deployments, and developers no longer need to actively watch deployments to production.

Common anti-patterns:

- You perform manual changes.
- You skip steps in your automation through manual emergency workflows.
- You don't follow your established plans and processes in favor of accelerated timelines.
- You perform rapid follow-on deployments without allowing for bake time.

Benefits of establishing this best practice: When you use automation to deploy all changes, you remove the potential for introduction of human error and provide the ability to test before you change production. Performing this process prior to production push verifies that your plans are complete. Additionally, automatic rollback into your release process can identify production issues and return your workload to its previously-working operational state.

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. An integral part of this is the adoption of the culture of continuous integration and continuous delivery/deployment (CI/CD), where a commit or code change passes through various automated stage gates from build and test stages to deployment on production environments.

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

Implementation steps

You can automate deployments to remove manual operations by following these steps:

- **Set up a code repository to store your code securely:** Use a hosted source code management system based on a popular technology such as Git to store your source code and infrastructure as code (IaC) configuration.

- **Configure a continuous integration service to compile your source code, run tests, and create deployment artifacts:** To set up a build project for this purpose, see Getting started with AWS CodeBuild using the console.
- **Set up a deployment service that automates application deployments and handles the complexity of application updates without reliance on error-prone manual deployments:** AWS CodeDeploy automates software deployments to a variety of compute services, such as Amazon EC2, AWS Fargate, AWS Lambda, and your on-premise servers. To configure these steps, see Getting started with CodeDeploy.
- **Set up a continuous delivery service that automates your release pipelines for quicker and more reliable application and infrastructure updates:** Consider using AWS CodePipeline to help you automate your release pipelines. For more detail, see CodePipeline tutorials.

Resources

Related best practices:

- OPS05-BP04 Use build and deployment management systems
- OPS05-BP10 Fully automate integration and deployment
- OPS06-BP02 Test deployments
- OPS06-BP04 Automate testing and rollback

Related documents:

- Continuous Delivery of Nested AWS CloudFormation Stacks Using AWS CodePipeline
- 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
- REL09-BP02 Secure and encrypt backups
- REL09-BP03 Perform data backup automatically
- REL09-BP04 Perform periodic recovery of the data to verify backup integrity and processes

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

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

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 allows recovery or the 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

All AWS data stores offer backup capabilities. Services such as Amazon RDS and Amazon DynamoDB additionally support automated backup that allows 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. AWS Elastic Disaster Recovery allows you to copy full server workloads and maintain continuous data protection from on-premise, cross-AZ or cross-Region, with a Recovery Point Objective (RPO) measured in seconds.

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 Amazon 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 Amazon EMR from Amazon 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.

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 allows creating backups of various data sources on AWS. AWS Elastic Disaster Recovery handles automated sub-second data replication to an AWS Region. Most AWS 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

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

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
- What is AWS Elastic Disaster Recovery?

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)

REL09-BP02 Secure and encrypt backups

Control and detect access to backups using authentication and authorization. Prevent and detect if data integrity of backups is compromised using encryption.

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

Control and detect access to backups using authentication and authorization, such as AWS Identity and Access Management (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. When using AWS Elastic Disaster Recovery, all data in transit and at rest is encrypted. With Elastic Disaster Recovery, data at rest can be encrypted using either the default Amazon EBS encryption Volume Encryption Key or a custom customer-managed key.

Implementation steps

1. Use encryption on each of your data stores. If your source data is encrypted, then the backup will also be encrypted.

- Use encryption in Amazon RDS.. You can configure encryption at rest using AWS Key Management Service when you create an RDS instance.
- Use encryption on Amazon EBS volumes.. You can configure default encryption or specify a unique key upon volume creation.
- 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.
- Encrypt your data stored in Amazon EFS. Configure the encryption when you create your file system.
- 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.
- Choose whether to use the default or custom Amazon EBS encryption for Elastic Disaster Recovery. This option will encrypt your replicated data at rest on the Staging Area Subnet disks and the replicated disks.

2. 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.

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 Amazon EFS
- Encryption for Backups in AWS
- Managing Encrypted Tables
- Security Pillar - AWS Well-Architected Framework

- What is AWS Elastic Disaster Recovery?

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.

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 verifies 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

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 turned on, 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.

AWS Elastic Disaster Recovery provides continuous block-level replication from the source environment (on-premises or AWS) to the target recovery region. Point-in-time Amazon EBS snapshots are automatically created and managed by the service.

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.

Implementation steps

1. **Identify data sources** that are currently being backed up manually. For more detail, see REL09-BP01 Identify and back up all data that needs to be backed up, or reproduce the data from sources.
2. **Determine the RPO** for the workload. For more detail, see REL13-BP01 Define recovery objectives for downtime and data loss.
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. Using backup plans in AWS Backup, create 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. For hands-on guidance on how to create automated backups using AWS Backup, see Testing Backup and Restore of Data. 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 invoke it at a frequency based on your RPO.

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?
- What is AWS Elastic Disaster Recovery?

Related videos:

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

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.

Desired outcome: Data from backups is periodically recovered using well-defined mechanisms to verify 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 check 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 when necessary, without using a runbook or outside of an established automated procedure.

Benefits of establishing this best practice: Testing the recovery of the backups verifies that 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 data is available, is not corrupted, is accessible, and that 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.

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.

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.

AWS Elastic Disaster Recovery offers continual point-in-time recovery snapshots of Amazon EBS volumes. As source servers are replicated, point-in-time states are chronicled over time based on the configured policy. Elastic Disaster Recovery helps you verify the integrity of these snapshots by launching instances for test and drill purposes without redirecting the traffic.

Implementation steps

1. **Identify data sources** that are currently being backed up and where these backups are being stored. For implementation guidance, see REL09-BP01 Identify and back up all data that needs to be backed up, or reproduce the data from sources.

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. For implementation guidance, see REL13-BP01 Define recovery objectives for downtime and data loss.
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 based on criteria you previously established for data validation. Does the restored and recovered data contain the most recent record or 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 your established RTO. 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 invoke 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.

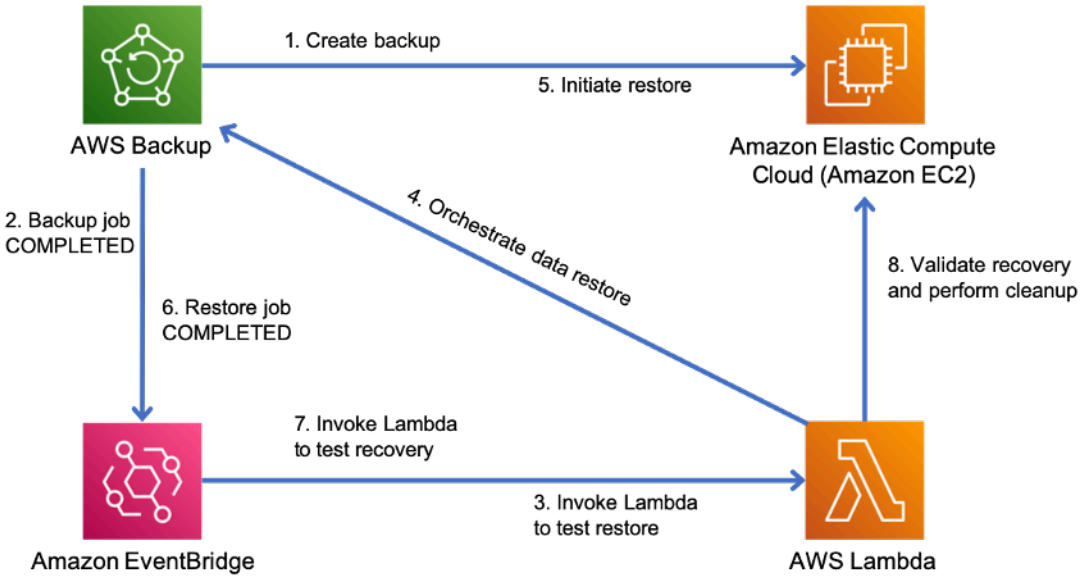


Figure 9. An automated backup and restore process

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

Use fault isolation to protect your workload

Fault isolation limits the impact of a component or system failure to a defined boundary. With proper isolation, components outside of the boundary are unaffected by the failure. Running your workload across multiple fault isolation boundaries can make it more resilient to failure.

Best practices

- REL10-BP01 Deploy the workload to multiple locations
- REL10-BP02 Automate recovery for components constrained to a single location
- REL10-BP03 Use bulkhead architectures to limit scope of impact

REL10-BP01 Deploy the workload to multiple locations

Distribute workload data and resources across multiple Availability Zones or, where necessary, across AWS Regions.

A fundamental principle for service design in AWS is to avoid single points of failure, including the underlying physical infrastructure. AWS provides cloud computing resources and services globally across multiple geographic locations called Regions. Each Region is physically and logically independent and consists of three or more Availability Zones (AZs). Availability Zones are geographically close to each other but are physically separated and isolated. When you distribute your workloads among Availability Zones and Regions, you mitigate the risk of threats such as fires, floods, weather-related disasters, earthquakes, and human error.

Create a location strategy to provide high availability that is appropriate for your workloads.

Desired outcome: Production workloads are distributed among multiple Availability Zones (AZs) or Regions to achieve fault tolerance and high availability.

Common anti-patterns:

- Your production workload exists only in a single Availability Zone.
- You implement a multi-Region architecture when a multi-AZ architecture would satisfy business requirements.
- Your deployments or data become desynchronized, which results in configuration drift or under-replicated data.
- You don't account for dependencies between application components if resilience and multi-location requirements differ between those components.

Benefits of establishing this best practice:

- Your workload is more resilient to incidents, such as power or environmental control failures, natural disasters, upstream service failures, or network issues that impact an AZ or an entire Region.
- You can access a wider inventory of Amazon EC2 instances and reduce the likelihood of InsufficientCapacityExceptions (ICE) when launching specific EC2 instance types.

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

Implementation guidance

Deploy and operate all production workloads in at least two Availability Zones (AZs) in a Region.

Using multiple Availability Zones

Availability Zones are resource hosting locations that are physically separated from each other to avoid correlated failures due to risks such as fires, floods, and tornadoes. Each Availability Zone has independent physical infrastructure, including utility power connections, backup power sources, mechanical services, and network connectivity. This arrangement limits faults in any of these components to just the impacted Availability Zone. For example, if an AZ-wide incident makes EC2 instances unavailable in the affected Availability Zone, your instances in other Availability Zone remains available.

Despite being physically separated, Availability Zones in the same AWS Region are close enough to provide high-throughput, low-latency (single-digit millisecond) networking. You can replicate data synchronously between Availability Zones for most workloads without significantly impacting user experience. This means you can use Availability Zones in a Region in an active/active or active/standby configuration.

All compute associated with your workload should be distributed among multiple Availability Zones. This includes Amazon EC2 instances, AWS Fargate tasks, and VPC-attached AWS Lambda functions. AWS compute services, including EC2 Auto Scaling, Amazon Elastic Container Service (ECS), and Amazon Elastic Kubernetes Service (EKS), provide ways for you to launch and manage compute across Availability Zones. Configure them to automatically replace compute as needed in a different Availability Zone to maintain availability. To direct traffic to available Availability Zones, place a load balancer in front of your compute, such as an Application Load Balancer or Network Load Balancer. AWS load balancers can reroute traffic to available instances in the event of an Availability Zone impairment.

You should also replicate data for your workload and make it available in multiple Availability Zones. Some AWS managed data services, such as Amazon S3, Amazon Elastic File Service (EFS), Amazon Aurora, Amazon DynamoDB, Amazon Simple Queue Service (SQS), and Amazon Kinesis Data Streams replicate data in multiple Availability Zones by default and are robust against Availability Zone impairment. With other AWS managed data services, such as Amazon Relational Database Service (RDS), Amazon Redshift, and Amazon ElastiCache, you must enable multi-AZ replication. Once enabled, these services automatically detect an Availability Zone impairment, redirect requests to an available Availability Zone, and re-replicate data as needed after recovery without customer intervention. Familiarize yourself with the user guide for each AWS managed data service you use to understand its multi-AZ capabilities, behaviors, and operations.

If you are using self-managed storage, such as Amazon Elastic Block Store (EBS) volumes or Amazon EC2 instance storage, you must manage multi-AZ replication yourself.

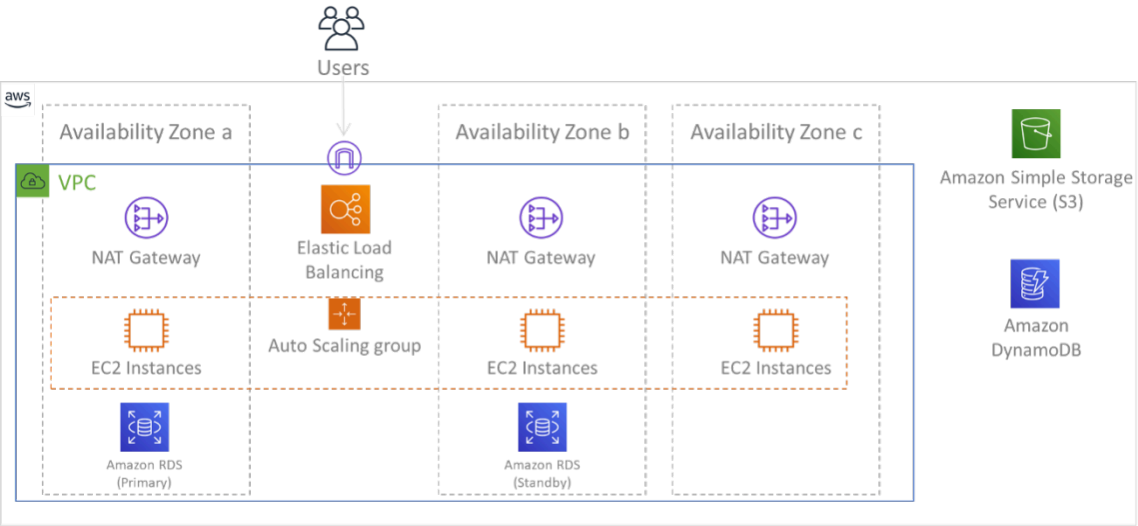


Figure 9: Multi-tier architecture deployed across three Availability Zones. Note that Amazon S3 and Amazon DynamoDB are always Multi-AZ automatically. The ELB also is deployed to all three zones.

Using multiple AWS Regions

If you have workloads that require extreme resilience (such as critical infrastructure, health-related applications, or services with stringent customer or mandated availability requirements), you may require additional availability beyond what a single AWS Region can provide. In this case, you should deploy and operate your workload across at least two AWS Regions (assuming that your data residency requirements allow it).

AWS Regions are located in different geographical regions around the world and in multiple continents. AWS Regions have even greater physical separation and isolation than Availability Zones alone. AWS services, with few exceptions, take advantage of this design to operate fully independently between different Regions (also known as *Regional services*). A failure of an AWS Regional service is designed not to impact the service in a different Region.

When you operate your workload in multiple Regions, you should consider additional requirements. Because resources in different Regions are separate from and independent of one another, you must duplicate your workload's components in each Region. This includes foundational infrastructure, such as VPCs, in addition to compute and data services.

NOTE: When you consider a multi-Regional design, verify that your workload is capable of running in a single Region. If you create dependencies between Regions where a component in one Region relies on services or components in a different Region, you can increase the risk of failure and significantly weaken your reliability posture.

To ease multi-Regional deployments and maintain consistency, AWS CloudFormation StackSets can replicate your entire AWS infrastructure across multiple Regions. AWS CloudFormation can also detect configuration drift and inform you when your AWS resources in a Region are out of sync. Many AWS services offer multi-region replication for important workload assets. For example, EC2 Image Builder can publish your EC2 machine images (AMIs) after every build to each Region you use. Amazon Elastic Container Registry (ECR) can replicate your container images to your selected Regions.

You must also replicate your data across each of your chosen Regions. Many AWS managed data services provide cross-Regional replication capability, including Amazon S3, Amazon DynamoDB, Amazon RDS, Amazon Aurora, Amazon Redshift, Amazon Elasticache, and Amazon EFS. Amazon DynamoDB global tables accept writes in any supported Region and will replicate data among all your other configured Regions. With other services, you must designate a primary Region for writes, as other Regions contain read-only replicas. For each AWS-managed data service your workload uses, refer to its user guide and developer guide to understand its multi-Region

capabilities and limitations. Pay special attention to where writes must be directed, transactional capabilities and limitations, how replication is performed, and how to monitor synchronization between Regions.

AWS also provides the ability to route request traffic to your Regional deployments with great flexibility. For example, you can configure your DNS records using Amazon Route 53 to direct traffic to the closest available Region to the user. Alternatively, you can configure your DNS records in an active/standby configuration, where you designate one Region as primary and fall back to a Regional replica only if the primary Region becomes unhealthy. You can configure Route 53 health checks to detect unhealthy endpoints and perform automatic failover and additionally use Amazon Application Recovery Controller (ARC) to provide a highly-available routing control for manually re-routing traffic as needed.

Even if you choose not to operate in multiple Regions for high availability, consider multiple Regions as part of your disaster recovery (DR) strategy. If possible, replicate your workload's infrastructure components and data in a *warm standby* or *pilot light* configuration in a secondary Region. In this design, you replicate baseline infrastructure from the primary Region such as VPCs, Auto Scaling groups, container orchestrators, and other components, but you configure the variable-sized components in the standby Region (such as the number of EC2 instances and database replicas) to be a minimally-operable size. You also arrange for continuous data replication from the primary Region to the standby Region. If an incident occurs, you can then scale out, or grow, the resources in the standby Region, and then promote it to become the primary Region.

Implementation steps

1. Work with business stakeholders and data residency experts to determine which AWS Regions can be used to host your resources and data.
2. Work with business and technical stakeholders to evaluate your workload, and determine whether its resilience needs can be met by a multi-AZ approach (single AWS Region) or if they require a multi-Region approach (if multiple Regions are permitted). The use of multiple Regions can achieve greater availability but can involve additional complexity and cost. Consider the following factors in your evaluation:
    a. **Business objectives and customer requirements**: How much downtime is permitted should a workload-impacting incident occur in an Availability Zone or a Region? Evaluate your recovery point objectives as discussed in REL13-BP01 Define recovery objectives for downtime and data loss.
    b. **Disaster recovery (DR) requirements**: What kind of potential disaster do you want to insure yourself against? Consider the possibility of data loss or long-term unavailability at

different scopes of impact from a single Availability Zone to an entire Region. If you replicate data and resources across Availability Zones, and a single Availability Zone experiences a sustained failure, you can recover service in another Availability Zone. If you replicate data and resources across Regions, you can recover service in another Region.

3. Deploy your compute resources into multiple Availability Zones.
    a. In your VPC, create multiple subnets in different Availability Zones. Configure each to be large enough to accommodate the resources needed to serve the workload, even during an incident. For more detail, see REL02-BP03 Ensure IP subnet allocation accounts for expansion and availability.
    b. If you are using Amazon EC2 instances, use EC2 Auto Scaling to manage your instances. Specify the subnets you chose in the previous step when you create your Auto Scaling groups.
    c. If you are using AWS Fargate compute for Amazon ECS or Amazon EKS, select the subnets you chose in the first step when you create an ECS Service, launch an ECS task, or create a Fargate profile for EKS.
    d. If you are using AWS Lambda functions that need to run in your VPC, select the subnets you chose in the first step when you create the Lambda function. For any functions that do not have a VPC configuration, AWS Lambda manages availability for you automatically.
    e. Place traffic directors such as load balancers in front of your compute resources. If cross-zone load balancing is enabled, AWS Application Load Balancers and Network Load Balancers detect when targets such as EC2 instances and containers are unreachable due to Availability Zone impairment and reroute traffic towards targets in healthy Availability Zones. If you disable cross-zone load balancing, use Amazon Application Recovery Controller (ARC) to provide zonal shift capability. If you are using a third-party load balancer or have implemented your own load balancers, configure them with multiple front ends across different Availability Zones.
4. Replicate your workload's data across multiple Availability Zones.
    a. If you use an AWS-managed data service such as Amazon RDS, Amazon ElastiCache, or Amazon FSx, study its user guide to understand its data replication and resilience capabilities. Enable cross-AZ replication and failover if necessary.
    b. If you use AWS-managed storage services such as Amazon S3, Amazon EFS, and Amazon FSx, avoid using single-AZ or One Zone configurations for data that requires high durability. Use a multi-AZ configuration for these services. Check the respective service's user guide to determine whether multi-AZ replication is enabled by default or whether you must enable it.

c. If you run a self-managed database, queue, or other storage service, arrange for multi-AZ replication according to the application's instructions or best practices. Familiarize yourself with the failover procedures for your application.

5. Configure your DNS service to detect AZ impairment and reroute traffic to a healthy Availability Zone. Amazon Route 53, when used in combination with Elastic Load Balancers, can do this automatically. Route 53 can also be configured with failover records that use health checks to respond to queries with only healthy IP addresses. For any DNS records used for failover, specify a short time to live (TTL) value (for example, 60 seconds or less) to help prevent record caching from impeding recovery (Route 53 alias records supply appropriate TTLs for you).

Additional steps when using multiple AWS Regions

1. Replicate all operating system (OS) and application code used by your workload across your selected Regions. Replicate Amazon Machine Images (AMIs) used by your EC2 instances if necessary using solutions such as Amazon EC2 Image Builder. Replicate container images stored in registries using solutions such as Amazon ECR cross-Region replication. Enable Regional replication for any Amazon S3 buckets used for storing application resources.
2. Deploy your compute resources and configuration metadata (such as parameters stored in AWS Systems Manager Parameter Store) into multiple Regions. Use the same procedures described in previous steps, but replicate the configuration for each Region you are using for your workload. Use infrastructure as code solutions such as AWS CloudFormation to uniformly reproduce the configurations among Regions. If you are using a secondary Region in a pilot light configuration for disaster recovery, you may reduce the number of your compute resources to a minimum value to save cost, with a corresponding increase in time to recovery.
3. Replicate your data from your primary Region into your secondary Regions.
   a. Amazon DynamoDB global tables provide global replicas of your data that can be written to from any supported Region. With other AWS-managed data services, such as Amazon RDS, Amazon Aurora, and Amazon Elasticache, you designate a primary (read/write) Region and replica (read-only) Regions. Consult the respective services' user and developer guides for details on Regional replication.
   b. If you are running a self-managed database, arrange for multi-Region replication according to the application's instructions or best practices. Familiarize yourself with the failover procedures for your application.

   c. If your workload uses AWS EventBridge, you may need to forward selected events from your primary Region to your secondary Regions. To do so, specify event buses in your secondary Regions as targets for matched events in your primary Region.

4. Consider whether and to what extent you want to use identical encryption keys across Regions. A typical approach that balances security and ease of use is to use Region-scoped keys for Region-local data and authentication, and use globally-scoped keys for encryption of data that is replicated among different Regions. AWS Key Management Service (KMS) supports multi-region keys to securely distribute and protect keys shared across Regions.
5. Consider AWS Global Accelerator to improve the availability of your application by directing traffic to Regions that contain healthy endpoints.

Resources

Related best practices:

- REL02-BP03 Ensure IP subnet allocation accounts for expansion and availability
- REL11-BP05 Use static stability to prevent bimodal behavior
- REL13-BP01 Define recovery objectives for downtime and data loss

Related documents:

- AWS Global Infrastructure
- White paper: AWS Fault Isolation Boundaries
- Resilience in Amazon EC2 Auto Scaling
- Amazon EC2 Auto Scaling: Example: Distribute instances across Availability Zones
- How EC2 Image Builder works
- How Amazon ECS places tasks on container instances (includes Fargate)
- Resilience in AWS Lambda
- Amazon S3: Replicating objects overview
- Private image replication in Amazon ECR
- Global Tables: Multi-Region Replication with DynamoDB
- Amazon Elasticache for Redis OSS: Replication across AWS Regions using global datastores
- Resilience in Amazon RDS

- Using Amazon Aurora global databases
- AWS Global Accelerator Developer Guide
- Multi-Region keys in AWS KMS
- Amazon Route 53: Configuring DNS failover
- Amazon Application Recovery Controller (ARC) Developer Guide
- Sending and receiving Amazon EventBridge events between AWS Regions
- Creating a Multi-Region Application with AWS Services blog series
- Disaster Recovery (DR) Architecture on AWS, Part I: Strategies for Recovery in the Cloud
- Disaster Recovery (DR) Architecture on AWS, Part III: Pilot Light and Warm Standby

Related videos:

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

REL10-BP02 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, implement the capability to do a complete rebuild of the workload within your defined recovery objectives.

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

Implementation guidance

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.

For stateful server-based workloads deployed to an on-premise data center, you can use AWS Elastic Disaster Recovery to protect your workloads in AWS. If you are already hosted in AWS, you can use Elastic Disaster Recovery to protect your workload to an alternative Availability Zone or Region. Elastic Disaster Recovery uses continual block-level replication to a lightweight staging area to provide fast, reliable recovery of on-premises and cloud-based applications.

Implementation steps

1. 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 Amazon EC2 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.
     - The launch template user data can be used to implement automation that can self-heal most workloads.
   - Use automatic recovery of Amazon EC2 instances for workloads that require a single instance ID address, private IP address, elastic IP address, and instance metadata.
     - Automatic Recovery will send recovery status alerts to a SNS topic as the instance failure is detected.
   - Use Amazon EC2 instance lifecycle events or Amazon ECS events to automate self-healing where automatic scaling or EC2 recovery cannot be used.
     - Use the events to invoke automation that will heal your component according to the process logic you require.
   - Protect stateful workloads that are limited to a single location using AWS Elastic Disaster Recovery.

Resources

Related documents:

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

REL10-BP03 Use bulkhead architectures to limit scope of impact

Implement bulkhead architectures (also known as cell-based architectures) to restrict the effect of failure within a workload to a limited number of components.

Desired outcome: A cell-based architecture uses multiple isolated instances of a workload, where each instance is known as a cell. Each cell is independent, does not share state with other cells, and handles a subset of the overall workload requests. This reduces the potential impact of a failure, such as a bad software update, to an individual cell and the requests it is processing. If a workload uses 10 cells to service 100 requests, when a failure occurs, 90% of the overall requests would be unaffected by the failure.

Common anti-patterns:

- Allowing cells to grow without bounds.
- Applying code updates or deployments to all cells at the same time.
- Sharing state or components between cells (with the exception of the router layer).
- Adding complex business or routing logic to the router layer.
- Not minimizing cross-cell interactions.

Benefits of establishing this best practice: With cell-based architectures, many common types of failure are contained within the cell itself, providing additional fault isolation. These fault boundaries can provide resilience against failure types that otherwise are hard to contain, such as unsuccessful code deployments or requests that are corrupted or invoke a specific failure mode (also known as *poison pill requests*).

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

Implementation guidance

On a ship, bulkheads ensure that a hull breach is contained within one section of the hull. In complex systems, this pattern is often replicated to allow fault isolation. Fault isolated boundaries restrict 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. On AWS, customers can use multiple Availability Zones and Regions to provide fault isolation, but the concept of fault isolation can be extended to your workload's architecture as well.

The overall workload is partitioned cells by a partition key. This key needs to align with the *grain* of the service, or the natural way that a service's workload can be subdivided with minimal cross-cell interactions. Examples of partition keys are customer ID, resource ID, or any other parameter easily accessible in most API calls. A cell routing layer distributes requests to individual cells based on the partition key and presents a single endpoint to clients.

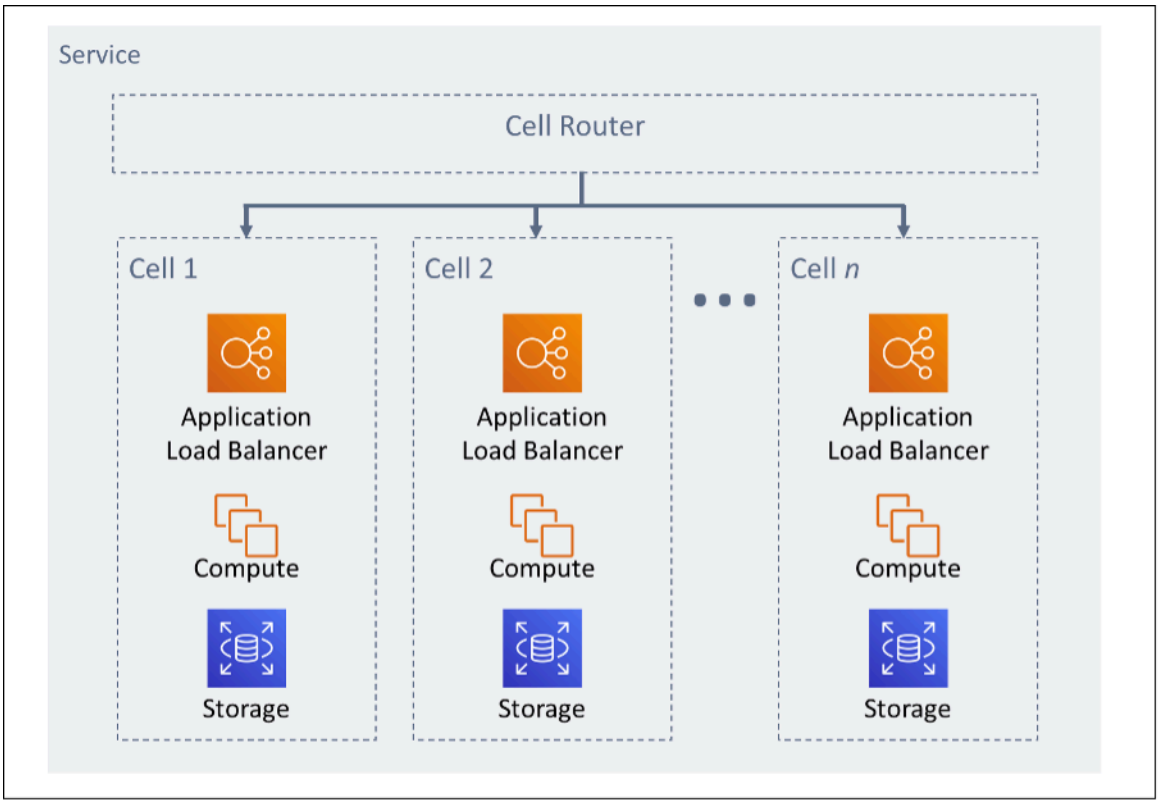


Figure 11: Cell-based architecture

Implementation steps

When designing a cell-based architecture, there are several design considerations to consider:

1. **Partition key**: Special consideration should be taken while choosing the partition key.
    - It should align with the grain of the service, or the natural way that a service's workload can be subdivided with minimal cross-cell interactions. Examples are `customer ID` or `resource ID`.
    - The partition key must be available in all requests, either directly or in a way that could be easily inferred deterministically by other parameters.
2. **Persistent cell mapping**: Upstream services should only interact with a single cell for the lifecycle of their resources.
    - Depending on the workload, a cell migration strategy may be needed to migrate data from one cell to another. A possible scenario when a cell migration may be needed is if a particular user or resource in your workload becomes too big and requires it to have a dedicated cell.
    - Cells should not share state or components between cells.
    - Consequently, cross-cell interactions should be avoided or kept to a minimum, as those interactions create dependencies between cells and therefore diminish the fault isolation improvements.
3. **Router layer**: The router layer is a shared component between cells, and therefore cannot follow the same compartmentalization strategy as with cells.
    - It is recommended for the router layer to distribute requests to individual cells using a partition mapping algorithm in a computationally efficient manner, such as combining cryptographic hash functions and modular arithmetic to map partition keys to cells.
    - To avoid multi-cell impacts, the routing layer must remain as simple and horizontally scalable as possible, which necessitates avoiding complex business logic within this layer. This has the added benefit of making it easy to understand its expected behavior at all times, allowing for thorough testability. As explained by Colm MacCárthaigh in Reliability, constant work, and a good cup of coffee, simple designs and constant work patterns produce reliable systems and reduce anti-fragility.
4. **Cell size**: Cells should have a maximum size and should not be allowed to grow beyond it.
    - The maximum size should be identified by performing thorough testing, until breaking points are reached and safe operating margins are established. For more detail on how to implement testing practices, see REL07-BP04 Load test your workload
    - The overall workload should grow by adding additional cells, allowing the workload to scale with increases in demand.
5. **Multi-AZ or Multi-Region strategies**: Multiple layers of resilience should be leveraged to protect against different failure domains.

- 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. 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. For more detail, see REL10-BP01 Deploy the workload to multiple locations.

6. **Code deployment**: A staggered code deployment strategy should be preferred over deploying code changes to all cells at the same time.
   - This helps minimize potential failure to multiple cells due to a bad deployment or human error. For more detail, see Automating safe, hands-off deployment.

Resources

Related best practices:

- REL07-BP04 Load test your workload
- REL10-BP01 Deploy the workload to multiple locations

Related documents:

- Reliability, constant work, and a good cup of coffee
- AWS and Compartmentalization
- Workload isolation using shuffle-sharding
- Automating safe, hands-off deployment

Related videos:

- AWS re:Invent 2018: Close Loops and Opening Minds: How to Take Control of Systems, Big and Small
- 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)
- AWS Summit ANZ 2021 - Everything fails, all the time: Designing for resilience

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
- REL11-BP02 Fail over to healthy resources
- REL11-BP03 Automate healing on all layers
- REL11-BP04 Rely on the data plane and not the control plane during recovery
- REL11-BP05 Use static stability to prevent bimodal behavior
- REL11-BP06 Send notifications when events impact availability
- REL11-BP07 Architect your product to meet availability targets and uptime service level agreements (SLAs)

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

Continually monitor the health of your workload so that you and your automated systems are aware of failures or degradations 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.

Desired outcome: Essential components of a workload are monitored independently to detect and alert on failures when and where they happen.

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 interfaces of the workload are actively monitored.
- Only collecting technical metrics, no business function metrics.

- No metrics measuring the user experience of the workload.
- Too many monitors are created.

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

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

Implementation guidance

Identify all workloads that will be reviewed for monitoring. Once you have identified all components of the workload that will need to monitored, you will now need to determine the monitoring interval. The monitoring interval will have a direct impact on how fast recovery can be initiated based on the time it takes to detect a failure. The mean time to detection (MTTD) is the amount of time between a failure occurring and when repair operations begin. The list of services should be extensive and complete.

Monitoring must cover all layers of the application stack including application, platform, infrastructure, and network.

Your monitoring strategy should consider the impact of *gray failures*. For more detail on gray failures, see Gray failures in the Advanced Multi-AZ Resilience Patterns whitepaper.

Implementation steps

- 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 and managed services.
  - Determine if detailed monitoring for EC2 instances and Auto Scaling is necessary. Detailed monitoring provides one minute interval metrics, and default monitoring provides five minute interval metrics.
  - Determine if enhanced monitoring for RDS is necessary. Enhanced monitoring uses an agent on RDS instances to get useful information about different process or threads.
  - Determine the monitoring requirements of critical serverless components for Lambda, API Gateway, Amazon EKS, Amazon ECS, and all types of load balancers.
  - Determine the monitoring requirements of storage components for Amazon S3, Amazon FSx, Amazon EFS, and Amazon EBS.

- Create custom metrics to measure business key performance indicators (KPIs). Workloads implement key business functions, which should be used as KPIs that help identify when an indirect problem happens.
- 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.
- 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.
- Set alarms to detect when any part of your workload is not working properly and to indicate when to automatically scale resources. Alarms can be visually displayed on dashboards, send alerts through Amazon SNS or email, and work with Auto Scaling to scale workload resources up or down.
- 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.
- Create distributed tracing monitoring for your services. With distributed monitoring, you can understand how your application and its underlying services are performing to identify and troubleshoot the root cause of performance issues and errors.
- Create monitoring systems (using CloudWatch or X-Ray) dashboards and data collection in a separate Region and account.
- Stay informed about service degradations with AWS Health. Create purpose-fit AWS Health event notifications to e-mail and chat channels through AWS User Notifications and integrate programmatically with your monitoring and alerting tools through Amazon EventBridge.

Resources

Related best practices:

- Availability Definition
- REL11-BP06 Send Notifications when events impact availability

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
- Using Cross Region Cross Account CloudWatch Dashboards
- Using Cross Region Cross Account X-Ray Tracing
- Understanding availability

Related videos:

- Mitigating gray failures

Related examples:

- One Observability Workshop: Explore X-Ray

Related tools:

- CloudWatch
- CloudWatch X-Ray

REL11-BP02 Fail over to healthy resources

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

When designing a service, distribute load across resources, Availability Zones, or Regions. Therefore, failure of an individual resource or impairment can be mitigated by shifting traffic to remaining healthy resources. Consider how services are discovered and routed to in the event of a failure.

Design your services with fault recovery in mind. At AWS, 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. 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.

The patterns and designs that allow for the failover vary for each AWS platform service. Many AWS native managed services are natively multiple Availability Zone (like Lambda or API Gateway). Other AWS services (like EC2 and EKS) require specific best practice designs to support failover of resources or data storage across AZs.

Monitoring should be set up to check that the failover resource is healthy, track the progress of the resources failing over, and monitor business process recovery.

Desired outcome: Systems are capable of automatically or manually using new resources to recover from degradation.

Common anti-patterns:

- Planning for failure is not part of the planning and design phase.
- RTO and RPO are not established.
- Insufficient monitoring to detect failing resources.
- Proper isolation of failure domains.
- Multi-Region fail over is not considered.
- Detection for failure is too sensitive or aggressive when deciding to failover.
- Not testing or validating failover design.
- Performing auto healing automation, but not notifying that healing was needed.
- Lack of dampening period to avoid failing back too soon.

Benefits of establishing this best practice: You can build more resilient systems that maintain reliability when experiencing failures by degrading gracefully and recovering quickly.

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

Implementation guidance

AWS services, such as Elastic Load Balancing and Amazon EC2 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, designs are more complicated. For example, cross-Region read replicas allow you to deploy your data to multiple AWS Regions. However, failover is still required to promote the read replica to primary and then point your traffic to the new endpoint. Amazon Route 53, Amazon Application Recovery Controller (ARC), Amazon CloudFront, and AWS Global Accelerator can help route traffic across AWS Regions.

AWS services, such as Amazon S3, Lambda, API Gateway, Amazon SQS, Amazon SNS, Amazon SES, Amazon Pinpoint, Amazon ECR, AWS Certificate Manager, EventBridge, or Amazon DynamoDB, are automatically deployed to multiple Availability Zones by AWS. In case of failure, these AWS services automatically route traffic to healthy locations. Data is redundantly stored in multiple Availability Zones and remains available.

For Amazon RDS, Amazon Aurora, Amazon Redshift, Amazon EKS, or Amazon ECS, Multi-AZ is a configuration option. AWS can direct traffic to the healthy instance if failover is initiated. This failover action may be taken by AWS or as required by the customer

For Amazon EC2 instances, Amazon Redshift, Amazon ECS tasks, or Amazon EKS pods, you choose which Availability Zones to deploy to. For some designs, Elastic Load Balancing provides the solution to detect instances in unhealthy zones and route traffic to the healthy ones. Elastic Load Balancing can also route traffic to components in your on-premises data center.

For Multi-Region traffic failover, rerouting can leverage Amazon Route 53, Amazon Application Recovery Controller, AWS Global Accelerator, Route 53 Private DNS for VPCs, or CloudFront to provide a way to define internet domains and assign routing policies, including health checks, to route traffic to healthy Regions. AWS Global Accelerator provides static IP addresses that act as a fixed entry point to your application, then route to endpoints in AWS Regions of your choosing, using the AWS global network instead of the internet for better performance and reliability.

Implementation steps

- Create failover designs for all appropriate applications and services. Isolate each architecture component and create failover designs meeting RTO and RPO for each component.

- Configure lower environments (like development or test) with all services that are required to have a failover plan. Deploy the solutions using infrastructure as code (IaC) to ensure repeatability.
- Configure a recovery site such as a second Region to implement and test the failover designs. If necessary, resources for testing can be configured temporarily to limit additional costs.
- Determine which failover plans are automated by AWS, which can be automated by a DevOps process, and which might be manual. Document and measure each service's RTO and RPO.
- Create a failover playbook and include all steps to failover each resource, application, and service.
- Create a failback playbook and include all steps to failback (with timing) each resource, application, and service
- Create a plan to initiate and rehearse the playbook. Use simulations and chaos testing to test the playbook steps and automation.
- For location impairment (such as Availability Zone or AWS Region), ensure you have systems in place to fail over to healthy resources in unimpaired locations. Check quota, autoscaling levels, and resources running before failover testing.

Resources

Related Well-Architected best practices:

- REL13- Plan for DR
- REL10 - Use fault isolation to protect your workload

Related documents:

- Setting RTO and RPO Targets
- Failover using Route 53 Weighted routing
- Disaster Recovery with Amazon Application Recovery Controller
- EC2 with autoscaling
- EC2 Deployments - Multi-AZ
- ECS Deployments - Multi-AZ
- Switch traffic using Amazon Application Recovery Controller

- Lambda with an Application Load Balancer and Failover
- ACM Replication and Failover
- Parameter Store Replication and Failover
- ECR cross region replication and Failover
- Secrets manager cross region replication configuration
- Enable cross region replication for EFS and Failover
- EFS Cross Region Replication and Failover
- Networking Failover
- S3 Endpoint failover using MRAP
- Create cross region replication for S3
- Guidance for Cross Region Failover and Graceful Failback on AWS
- Failover using multi-region global accelerator
- Failover with DRS

Related examples:

- Disaster Recovery on AWS
- Elastic Disaster Recovery on AWS

REL11-BP03 Automate healing on all layers

Upon detection of a failure, use automated capabilities to perform actions to remediate. Degradations may be automatically healed through internal service mechanisms or require resources to be restarted or removed through remediation actions.

For self-managed applications and cross-Region healing, recovery designs and automated healing processes can be pulled from existing best practices.

The ability to restart or remove a resource is an important tool to remediate failures. A best practice is to make services stateless where possible. This prevents loss of data or availability on resource restart. In the cloud, you can (and generally should) replace the entire resource (for example, a compute instance or serverless 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.

Restarting or retrying also applies to network requests. 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.

Desired outcome: Automated actions are performed to remediate detection of a failure.

Common anti-patterns:

- Provisioning resources without autoscaling.
- 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.
- No automation to failover databases.
- Lack automated methods to reroute traffic to new endpoints.
- No storage replication.

Benefits of establishing this best practice: Automated healing can reduce your mean time to recovery and improve your availability.

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

Implementation guidance

Designs for Amazon EKS or other Kubernetes services should include both minimum and maximum replica or stateful sets and the minimum cluster and node group sizing. These mechanisms provide a minimum amount of continually-available processing resources while automatically remediating any failures using the Kubernetes control plane.

Design patterns that are accessed through a load balancer using compute clusters should leverage Auto Scaling groups. Elastic Load Balancing (ELB) automatically distributes incoming application traffic across multiple targets and virtual appliances in one or more Availability Zones (AZs).

Clustered compute-based designs that do not use load balancing should have their size designed for loss of at least one node. This will allow for the service to maintain itself running in potentially

reduced capacity while it's recovering a new node. Example services are Mongo, DynamoDB Accelerator, Amazon Redshift, Amazon EMR, Cassandra, Kafka, MSK-EC2, Couchbase, ELK, and Amazon OpenSearch Service. Many of these services can be designed with additional auto healing features. Some cluster technologies must generate an alert upon the loss a node triggering an automated or manual workflow to recreate a new node. This workflow can be automated using AWS Systems Manager to remediate issues quickly.

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 invoke AWS Lambda, Systems Manager Automation, or other targets to run 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. For large-scale replacements (such as the loss of an entire Availability Zone), static stability is preferred for high availability.

Implementation steps

- Use Auto Scaling groups to deploy tiers in a workload. Auto Scaling can perform self-healing on stateless applications and add or remove capacity.
- For compute instances noted previously, use load balancing and choose the appropriate type of load balancer.
- Consider healing for Amazon RDS. With standby instances, configure for auto failover to the standby instance. For Amazon RDS Read Replica, automated workflow is required to make a read replica primary.
- 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 EBS volumes and mount points to Amazon Elastic File System or File Systems for Lustre and Windows. Using AWS OpsWorks, you can configure automatic healing of EC2 instances at the layer level.
- 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.

- 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 invoke AWS Lambda (or other targets) to run custom remediation logic on your workload.

Resources

Related best practices:

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

Related documents:

- How AWS Auto Scaling Works
- 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?
- AWS OpsWorks: Using Auto Healing to Replace Failed Instances
- What is AWS Step Functions?
- What is AWS Lambda?
- What Is Amazon EventBridge?
- Using Amazon CloudWatch Alarms
- Amazon RDS Failover
- SSM - Systems Manager Automation
- Resilient Architecture Best Practices

Related videos:

- Automatically Provision and Scale OpenSearch Service
- Amazon RDS Failover Automatically

Related examples:

- Amazon RDS Failover Workshop

Related tools:

- CloudWatch
- CloudWatch X-Ray

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

Control planes provide the administrative APIs used to create, read and describe, update, delete, and list (CRUDL) resources, while data planes handle day-to-day service traffic. When implementing recovery or mitigation responses to potentially resiliency-impacting events, focus on using a minimal number of control plane operations to recover, rescale, restore, heal, or failover the service. Data plane action should supersede any activity during these degradation events.

For example, the following are all control plane actions: launching a new compute instance, creating block storage, and describing queue services. When you launch compute instances, the control plane has to perform multiple tasks like finding a physical host with capacity, allocating network interfaces, preparing local block storage volumes, generating credentials, and adding security rules. Control planes tend to be complicated orchestration.

Desired outcome: When a resource enters an impaired state, the system is capable of automatically or manually recovering by shifting traffic from impaired to healthy resources.

Common anti-patterns:

- Dependence on changing DNS records to re-route traffic.
- Dependence on control-plane scaling operations to replace impaired components due to insufficiently provisioned resources.
- Relying on extensive, multi service, multi-API control plane actions to remediate any category of impairment.

Benefits of establishing this best practice: Increased success rate for automated remediation can reduce your mean time to recovery and improve availability of the workload.

Level of risk exposed if this best practice is not established: Medium: For certain types of service degradations, control planes are affected. Dependencies on extensive use of the control plane for remediation may increase recovery time (RTO) and mean time to recovery (MTTR).

Implementation guidance

To limit data plane actions, assess each service for what actions are required to restore service.

Leverage Amazon Application Recovery Controller to shift the DNS traffic. These features continually monitor your application's ability to recover from failures and allow you to control your application recovery across multiple AWS Regions, Availability Zones, and on premises.

Route 53 routing policies use 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.

Design your compute infrastructure to be statically stable to avoid using the control plane during an incident. For example, if you are using Amazon EC2 instances, avoid provisioning new instances manually or instructing Auto Scaling Groups to add instances in response. For the highest levels of resilience, provision sufficient capacity in the cluster used for failover. If this capacity threshold must be limited, set throttles on the overall end-to-end system to safely limit the total traffic reaching the limited set of resources.

For services like Amazon DynamoDB, Amazon API Gateway, load balancers, and AWS Lambda serverless, using those services leverages the data plane. However, creating new functions, load balancers, API gateways, or DynamoDB tables is a control plane action and should be completed before the degradation as preparation for an event and rehearsal of failover actions. For Amazon RDS, data plane actions allow for access to data.

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

Understand which operations are on the data plane and which are on the control plane.

Implementation steps

For each workload that needs to be restored after a degradation event, evaluate the failover runbook, high availability design, auto healing design, or HA resource restoration plan. Identity each action that might be considered a control plane action.

Consider changing the control action to a data plane action:

- Auto Scaling (control plane) to pre-scaled Amazon EC2 resources (data plane)
- Amazon EC2 instance scaling (control plane) to AWS Lambda scaling (data plane)
- Assess any designs using Kubernetes and the nature of the control plane actions. Adding pods is a data plane action in Kubernetes. Actions should be limited to adding pods and not adding nodes. Using over-provisioned nodes is the preferred method to limit control plane actions

Consider alternate approaches that allow for data plane actions to affect the same remediation.

- Route 53 Record change (control plane) or Amazon Application Recovery Controller (data plane)
- Route 53 Health checks for more automated updates

Consider some services in a secondary Region, if the service is mission critical, to allow for more control plane and data plane actions in an unaffected Region.

- Amazon EC2 Auto Scaling or Amazon EKS in a primary Region compared to Amazon EC2 Auto Scaling or Amazon EKS in a secondary Region and routing traffic to secondary Region (control plane action)
- Make read replica in secondary primary or attempting same action in primary Region (control plane action)

Resources

Related best practices:

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

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 Application Recovery Controller, Part 1: Single-Region stack
- Building highly resilient applications using Amazon Application Recovery Controller, Part 2: Multi-Region stack
- Creating Disaster Recovery Mechanisms Using Amazon Route 53
- What is Amazon Application Recovery Controller
- Kubernetes Control Plane and data plane

Related videos:

- Back to Basics - Using Static Stability
- Building resilient multi-site workloads using AWS global services

Related examples:

- Introducing Amazon Application Recovery Controller
- Amazon Builders' Library: Avoiding overload in distributed systems by putting the smaller service in control
- Building highly resilient applications using Amazon Application Recovery Controller, Part 1: Single-Region stack
- Building highly resilient applications using Amazon Application Recovery Controller, Part 2: Multi-Region stack
- Static stability using Availability Zones

Related tools:

- Amazon CloudWatch
- AWS X-Ray

REL11-BP05 Use static stability to prevent bimodal behavior

Workloads should be statically stable and only operate in a single normal mode. Bimodal behavior is when your workload exhibits different behavior under normal and failure modes.

For example, you might try and recover from an Availability Zone failure by launching new instances in a different Availability Zone. This can result in a bimodal response during a failure mode. You should instead build workloads that are statically stable and operate within only one mode. In this example, those instances should have been provisioned in the second Availability Zone before the failure. This static stability design verifies that the workload only operates in a single mode.

Desired outcome: Workloads do not exhibit bimodal behavior during normal and failure modes.

Common anti-patterns:

- Assuming resources can always be provisioned regardless of the failure scope.
- Trying to dynamically acquire resources during a failure.
- Not provisioning adequate resources across zones or Regions until a failure occurs.
- Considering static stable designs for compute resources only.

Benefits of establishing this best practice: Workloads running with statically stable designs are capable of having predictable outcomes during normal and failure events.

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

Implementation guidance

Bimodal behavior occurs when your workload exhibits different behavior under normal and failure modes (for example, relying on launching new instances if an Availability Zone fails). An example of bimodal behavior is when stable Amazon EC2 designs provision enough instances in each Availability Zone to handle the workload load if one AZ were removed. Elastic Load Balancing or Amazon Route 53 health would check to shift a load away from the impaired instances. After traffic has shifted, use AWS Auto Scaling to asynchronously replace instances from the failed zone and

launch them in the healthy zones. Static stability for compute deployment (such as EC2 instances or containers) results in the highest reliability.

Static stability of EC2 instances across Availability Zones

This must be weighed against the cost for this model and the business value of maintaining the workload under all resilience cases. 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 or Regional impairment), this approach is less effective because it relies on both an operational plane, and sufficient resources being available in the unaffected zones or Regions.

Your solution should also weigh reliability against the costs needs for your workload. Static stability architectures apply to a variety of architectures including compute instances spread across Availability Zones, database read replica designs, Kubernetes (Amazon EKS) cluster designs, and multi-Region failover architectures.

It is also possible to implement a more statically stable design by using more resources in each zone. By adding more zones, you reduce the amount of additional compute you need for static stability.

An 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 an unexpected load to another component and might cause it to fail, resulting in other unexpected consequences. This negative feedback loop impacts the availability of your workload. Instead, you can build systems that are statically stable and operate in only one mode. A statically stable design would do constant work and always refresh the configuration state on a fixed cadence. When a call fails, the workload would use the previously cached value and initiate 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 it can significantly change the demands on your workload and is likely to result in failures.

Assess critical workloads to determine what workloads require this type of resilience design. For those that are deemed critical, each application component must be reviewed. Example types of services that require static stability evaluations are:

- **Compute**: Amazon EC2, EKS-EC2, ECS-EC2, EMR-EC2
- **Databases**: Amazon Redshift, Amazon RDS, Amazon Aurora
- **Storage**: Amazon S3 (Single Zone), Amazon EFS (mounts), Amazon FSx (mounts)
- **Load balancers:** Under certain designs

Implementation steps

- Build systems that are statically stable and operate in only one mode. In this case, provision enough instances in each Availability Zone or Region to handle the workload capacity if one Availability Zone or Region were removed. A variety of services can be used for routing to healthy resources, such as:
  - Cross Region DNS Routing
  - MRAP Amazon S3 MultiRegion Routing
  - AWS Global Accelerator
  - Amazon Application Recovery Controller
- Configure database read replicas to account for the loss of a single primary instance or a read replica. If traffic is being served by read replicas, the quantity in each Availability Zone and each Region should equate to the overall need in case of the zone or Region failure.
- Configure critical data in Amazon S3 storage that is designed to be statically stable for data stored in case of an Availability Zone failure. If Amazon S3 One Zone-IA storage class is used, this should not be considered statically stable, as the loss of that zone minimizes access to this stored data.
- Load balancers are sometimes configured incorrectly or by design to service a specific Availability Zone. In this case, the statically stable design might be to spread a workload across multiple AZs in a more complex design. The original design may be used to reduce interzone traffic for security, latency, or cost reasons.

Resources

Related Well-Architected best practices:

- Availability Definition
- REL11-BP01 Monitor all components of the workload to detect failures
- REL11-BP04 Rely on the data plane and not the control plane during recovery

Related documents:

- Minimizing Dependencies in a Disaster Recovery Plan
- The Amazon Builders' Library: Static stability using Availability Zones
- Fault Isolation Boundaries
- Static stability using Availability Zones
- Multi-Zone RDS
- Minimizing Dependencies in a Disaster Recovery Plan
- Cross Region DNS Routing
- MRAP Amazon S3 MultiRegion Routing
- AWS Global Accelerator
- Amazon Application Recovery Controller
- Single Zone Amazon S3
- Cross Zone Load Balancing

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 thresholds breached, even if the event causing the issue was automatically resolved.

Automated healing allows 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.

Resilient systems are designed so that degradation events are immediately communicated to the appropriate teams. These notifications should be sent through one or many communication channels.

Desired outcome: Alerts are immediately sent to operations teams when thresholds are breached, such as error rates, latency, or other critical key performance indicator (KPI) metrics, so that these issues are resolved as soon as possible and user impact is avoided or minimized.

Common anti-patterns:

- Sending too many alarms.
- Sending alarms that are not actionable.
- Setting alarm thresholds too high (over sensitive) or too low (under sensitive).
- Not sending alarms for external dependencies.
- Not considering gray failures when designing monitoring and alarms.
- Performing healing automation, but not notifying the appropriate team that healing was needed.

Benefits of establishing this best practice: Notifications of recovery make operational and business teams aware of service degradations so that they can react immediately to minimize both mean time to detect (MTTD) and mean time to repair (MTTR). Notifications of recovery events also assure that you don't ignore problems that occur infrequently.

Level of risk exposed if this best practice is not established: Medium. Failure to implement appropriate monitoring and events notification mechanisms can result in failure to detect patterns of problems, including those addressed by auto healing. A team will only be made aware of system degradation when users contact customer service or by chance.

Implementation guidance

When defining a monitoring strategy, a triggered alarm is a common event. This event would likely contain an identifier for the alarm, the alarm state (such as `IN ALARM` or `OK`), and details of what triggered it. In many cases, an alarm event should be detected and an email notification sent. This is an example of an action on an alarm. Alarm notification is critical in observability, as it informs the right people that there is an issue. However, when action on events mature in

your observability solution, it can automatically remediate the issue without the need for human intervention.

Once KPI-monitoring alarms have been established, alerts should be sent to appropriate teams when thresholds are exceeded. Those alerts may also be used to trigger automated processes that will attempt to remediate the degradation.

For more complex threshold monitoring, composite alarms should be considered. Composite alarms use a number of KPI-monitoring alarms to create an alert based on operational business logic. CloudWatch Alarms can be configured to send emails, or to log incidents in third-party incident tracking systems using Amazon SNS integration or Amazon EventBridge.

Implementation steps

Create various types of alarms based on how the workloads are monitored, such as:

- Application alarms are used to detect when any part of your workload is not working properly.
- Infrastructure alarms indicate when to scale resources. Alarms can be visually displayed on dashboards, send alerts through Amazon SNS or email, and work with Auto Scaling to scale workload resources in or out.
- Simple static alarms can be created to monitor when a metric breaches a static threshold for a specified number of evaluation periods.
- Composite alarms can account for complex alarms from multiple sources.
- Once the alarm has been created, create appropriate notification events. You can directly invoke an Amazon SNS API to send notifications and link any automation for remediation or communication.
- Stay informed about service degradations with AWS Health. Create purpose-fit AWS Health event notifications to e-mail and chat channels through AWS User Notifications and integrate programmatically with your monitoring and alerting tools through Amazon EventBridge.

Resources

Related Well-Architected best practices:

- Availability Definition

Related documents:

- Creating a CloudWatch Alarm Based on a Static Threshold
- What Is Amazon EventBridge?
- What is Amazon Simple Notification Service?
- Publishing Custom Metrics
- Using Amazon CloudWatch Alarms
- Setup CloudWatch Composite alarms
- What's new in AWS Observability at re:Invent 2022

Related tools:

- CloudWatch
- CloudWatch X-Ray

REL11-BP07 Architect your product to meet availability targets and uptime service level agreements (SLAs)

Architect your product to meet availability targets and uptime service level agreements (SLAs). If you publish or privately agree to availability targets or uptime SLAs, verify that your architecture and operational processes are designed to support them.

Desired outcome: Each application has a defined target for availability and SLA for performance metrics, which can be monitored and maintained in order to meet business outcomes.

Common anti-patterns:

- Designing and deploying workload's without setting any SLAs.
- SLA metrics are set too high without rationale or business requirements.
- Setting SLAs without taking into account for dependencies and their underlying SLA.
- Application designs are created without considering the Shared Responsibility Model for Resilience.

Benefits of establishing this best practice: Designing applications based on key resiliency targets helps you meet business objectives and customer expectations. These objectives help drive the application design process that evaluates different technologies and considers various tradeoffs.

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

Implementation guidance

Application designs have to account for a diverse set of requirements that are derived from business, operational, and financial objectives. Within the operational requirements, workloads need to have specific resilience metric targets so they can be properly monitored and supported. Resilience metrics should not be set or derived after deploying the workload. They should be defined during the design phase and help guide various decisions and tradeoffs.

- Every workload should have its own set of resilience metrics. Those metrics may be different from other business applications.
- Reducing dependencies can have a positive impact on availability. Each workload should consider its dependencies and their SLAs. In general, select dependencies with availability goals equal to or greater than the goals of your workload.
- Consider loosely coupled designs so your workload can operate correctly despite dependency impairment, where possible.
- Reduce control plane dependencies, especially during recovery or a degradation. Evaluate designs that are statically stable for mission critical workloads. Use resource sparing to increase the availability of those dependencies in a workload.
- Observability and instrumentation are critical for achieving SLAs by reducing Mean Time to Detection (MTTD) and Mean Time to Repair (MTTR).
- Less frequent failure (longer MTBF), shorter failure detection times (shorter MTTD), and shorter repair times (shorter MTTR) are the three factors that are used to improve availability in distributed systems.
- Establishing and meeting resilience metrics for a workload is foundational to any effective design. Those designs must factor in tradeoffs of design complexity, service dependencies, performance, scaling, and costs.

Implementation steps

- Review and document the workload design considering the following questions:
  - Where are control planes used in the workload?
  - How does the workload implement fault tolerance?
  - What are the design patterns for scaling, automatic scaling, redundancy, and highly available components?

  - What are the requirements for data consistency and availability?
  - Are there considerations for resource sparing or resource static stability?
  - What are the service dependencies?
- Define SLA metrics based on the workload architecture while working with stakeholders. Consider the SLAs of all dependencies used by the workload.
- Once the SLA target has been set, optimize the architecture to meet the SLA.
- Once the design is set that will meet the SLA, implement operational changes, process automation, and runbooks that also will have focus on reducing MTTD and MTTR.
- Once deployed, monitor and report on the SLA.

Resources

Related best practices:

- REL03-BP01 Choose how to segment your workload
- REL10-BP01 Deploy the workload to multiple locations
- REL11-BP01 Monitor all components of the workload to detect failures
- REL11-BP03 Automate healing on all layers
- REL12-BP04 Test resiliency using chaos engineering
- REL13-BP01 Define recovery objectives for downtime and data loss
- Understanding workload health

Related documents:

- Availability with redundancy
- Reliability pillar - Availability
- Measuring availability
- AWS Fault Isolation Boundaries
- Shared Responsibility Model for Resiliency
- Static stability using Availability Zones
- AWS Service Level Agreements (SLAs)
- Guidance for Cell-based Architecture on AWS

- AWS infrastructure
- Advanced Multi-AZ Resilience Patterns whitepaper

Related services:

- Amazon CloudWatch
- AWS Config
- AWS Trusted Advisor

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
- REL12-BP02 Perform post-incident analysis
- REL12-BP03 Test scalability and performance requirements
- REL12-BP04 Test resiliency using chaos engineering
- REL12-BP05 Conduct game days regularly

REL12-BP01 Use playbooks to investigate failures

Permit 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. Allow 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 invoked or performed as part of playbook activities, or might prompt to run 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.

Desired outcome: Your teams have a consistent and agreed upon approach to handling post-incident analysis. One mechanism is the correction of error (COE) process. The COE process helps your teams identify, understand, and address the root causes for incidents, while also building mechanisms and guardrails to limit the probability of the same incident happening again.

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.
- Focus on assigning blame rather than understanding the root cause, creating a culture of fear and hindering open communication
- Failure to share insights, which keeps incident analysis findings within a small group and prevents others from benefiting from the lessons learned
- No mechanism to capture institutional knowledge, thereby losing valuable insights by not preserving the lessons-learned in the form of updated best practices and resulting in repeat incidents with the same or similar root cause

Benefits of establishing this best practice: Conducting post-incident analysis and sharing the results permits other workloads to mitigate the risk if they have implemented the same contributing factors, and allows 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

Good post-incident analysis provides opportunities to propose common solutions for problems with architecture patterns that are used in other places in your systems.

A cornerstone of the COE process is documenting and addressing issues. It is recommended to define a standardized way to document critical root causes, and ensure they are reviewed and addressed. Assign clear ownership for the post-incident analysis process. Designate a responsible team or individual who will oversee incident investigations and follow-ups.

Encourage a culture that focuses on learning and improvement rather than assigning blame. Emphasize that the goal is to prevent future incidents, not to penalize individuals.

Develop well-defined procedures for conducting post-incident analyses. These procedures should outline the steps to be taken, the information to be collected, and the key questions to be addressed during the analysis. Investigate incidents thoroughly, going beyond immediate causes to

identify root causes and contributing factors. Use techniques like the *five whys* to delve deep into the underlying issues.

Maintain a repository of lessons learned from incident analyses. This institutional knowledge can serve as a reference for future incidents and prevention efforts. Share findings and insights from post-incident analyses, and consider holding open-invite post-incident review meetings to discuss lessons learned.

Implementation steps

- While conducting post-incident analysis, ensure the process is blame-free. This allows people involved in the incident to be dispassionate about the proposed corrective actions and promote honest self-assessment and collaboration across teams.
- Define a standardized way to document critical issues. An example structure for such document is as follows:
  - What happened?
  - What was the impact on customers and your business?
  - What was the root cause?
  - What data do you have to support this?
    - For example, metrics and graphs
  - What were the critical pillar implications, especially security?
    - When architecting workloads, you make trade-offs between pillars based upon your business context. These business decisions can drive your engineering priorities. You might optimize to reduce cost at the expense of reliability in development environments, or, for mission-critical solutions, you might optimize reliability with increased costs. Security is always job zero, as you have to protect your customers.
  - What lessons did you learn?
  - What corrective actions are you taking?
    - Action items
    - Related items
- Create well-defined standard operating procedures for conducting post-incident analyses.
- Set up a standardized incident reporting process. Document all incidents comprehensively, including the initial incident report, logs, communications, and actions taken during the incident.
- Remember that an incident does not require an outage. It could be a near-miss, or a system performing in an unexpected way while still fulfilling its business function.

- Continually improve your post-incident analysis process based on feedback and lessons learned.
- Capture key findings in a knowledge management system, and consider any patterns that should be added to developer guides or pre-deployment checklists.

Resources

Related documents:

- Why you should develop a correction of error (COE)

Related videos:

- Amazon's approach to failing successfully
- AWS re:Invent 2021 - Amazon Builders' Library: Operational Excellence at Amazon

REL12-BP03 Test scalability 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 for your workload on demand. Instead of reliance on a scaled-down test environment, which could lead to inaccurate predictions of production behaviors, you can use the cloud to provision a test environment that closely mirrors your expected production environment. This environment helps you test in a more accurate simulation of the real-world conditions your application faces.

Alongside your performance testing efforts, it's essential to validate that your base resources, scaling settings, service quotas, and resiliency design operate as expected under load. This holistic approach verifies that your application can reliably scale and perform as required, even under the most demanding conditions.

Desired outcome: Your workload maintains its expected behavior even while subject to peak load. You proactively address any performance-related issues that may arise as the application grows and evolves.

Common anti-patterns:

- You use test environments that do not closely match the production environment.

- You treat load testing as a separate, one-time activity rather than an integrated part of the deployment continuous integration (CI) pipeline.
- You don't define clear and measurable performance requirements, such as response time, throughput, and scalability targets.
- You perform tests with unrealistic or insufficient load scenarios, and you fail to test for peak loads, sudden spikes, and sustained high load.
- You don't stress test the workload by exceeding expected load limits.
- You use inadequate or inappropriate load testing and performance profiling tools.
- You lack comprehensive monitoring and alerting systems to track performance metrics and detect anomalies.

Benefits of establishing this best practice:

- Load testing helps you identify potential performance bottlenecks in your system before it goes into production. When you simulate production-level traffic and workloads, you can identify areas where your system may struggle to handle the load, such as slow response times, resource constraints, or system failures.
- As you test your system under various load conditions, you can better understand the resource requirements needed to support your workload. This information can help you make informed decisions about resource allocation and prevent over-provisioning or under-provisioning of resources.
- To identify potential failure points, you can observe how your workload performs under high load conditions. This information helps you improve your workload's reliability and resiliency by implementing fault-tolerance mechanisms, failover strategies, and redundancy measures, as appropriate.
- You identify and address performance issues early, which helps you avoid the costly consequences of system outages, slow response times, and dissatisfied users.
- Detailed performance data and profiling information collected during testing can help you troubleshoot performance-related issues that may arise in production. This can lead to faster incident response and resolution, which reduces the impact on users and your organization's operations.
- In certain industries, proactive performance testing can help your workload meet compliance standards, which reduces the risk of penalties or legal issues.

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

Implementation guidance

The first step is to define a comprehensive testing strategy that covers all aspects of scaling and performance requirements. To start, clearly define your workload's service-level objectives (SLOs) based on your business needs, such as throughput, latency histogram, and error rate. Next, design a suite of tests that can simulate various load scenarios that range from average usage to sudden spikes and sustained peak loads, and verify that the workload's behavior meets your SLOs. These tests should be automated and integrated into your continuous integration and deployment pipeline to catch performance regressions early in the development process.

To effectively test scaling and performance, invest in the right tools and infrastructure. This includes load testing tools that can generate realistic user traffic, performance profiling tools to identify bottlenecks, and monitoring solutions to track key metrics. Importantly, you should verify that your test environments closely match the production environment in terms of infrastructure and environment conditions to make your test results as accurate as possible. To make it easier to reliably replicate and scale production-like setups, use infrastructure as code and container-based applications.

Scaling and performance tests are an ongoing process, not a one-time activity. Implement comprehensive monitoring and alerting to track the application's performance in production, and use this data to continually refine your test strategies and optimization efforts. Regularly analyze performance data to identify emerging issues, test new scaling strategies, and implement optimizations to improve the application's efficiency and reliability. When you adopt an iterative approach and constantly learn from production data, you can verify that your application can adapt to variable user demands and maintain resiliency and optimal performance over time.

Implementation steps

1. Establish clear and measurable performance requirements, such as response time, throughput, and scalability targets. These requirements should be based on your workload's usage patterns, user expectations, and business needs.
2. Select and configure a load testing tool that can accurately mimic the load patterns and user behavior in your production environment.
3. Set up a test environment that closely matches the production environment, including infrastructure and environment conditions, to improve the accuracy of your test results.

4. Create a test suite that covers a wide range of scenarios, from average usage patterns to peak loads, rapid spikes, and sustained high loads. Integrate the tests into your continuous integration and deployment pipelines to catch performance regressions early in the development process.
5. Conduct load testing to simulate real-world user traffic and understand how your application behaves under different load conditions. To stress test your application, exceed the expected load and observe its behavior, such as response time degradation, resource exhaustion, or system failures, which helps identify the breaking point of your application and inform scaling strategies. Evaluate the scalability of your workload by incrementally increasing the load, and measure the performance impact to identify scaling limits and plan for future capacity needs.
6. Implement comprehensive monitoring and alerting to track performance metrics, detect anomalies, and initiate scaling actions or notifications when thresholds are exceeded.
7. Continually monitor and analyze performance data to identify areas for improvement. Iterate on your testing strategies and optimization efforts.

Resources

Related best practices:

- REL01-BP04 Monitor and manage quotas
- REL06-BP01 Monitor all components for the workload (Generation)
- REL06-BP03 Send notifications (Real-time processing and alarming))

Related documents:

- Load testing applications
- Distributed Load Testing on AWS
- Application Performance Monitoring
- Amazon EC2 Testing Policy

Related examples:

- Distributed Load Testing on AWS (GitHub)

Related tools:

- Amazon CodeGuru Profiler
- Amazon CloudWatch RUM
- Apache JMeter
- K6
- Vegeta
- Hey
- ab
- wrk
- Distributed Load Testing on AWS

REL12-BP04 Test resiliency using chaos engineering

Run chaos experiments regularly in environments that are in or as close to production as possible to understand how your system responds to adverse conditions.

Desired outcome:

The resilience of the workload is regularly verified by applying chaos engineering in the form of fault injection experiments or injection of unexpected load, in addition to resilience testing that validates known expected behavior of your workload during an event. Combine both chaos engineering and resilience testing to gain confidence that your workload can survive component failure and can recover from unexpected disruptions with minimal to no impact.

Common anti-patterns:

- Designing for resiliency, but not verifying how the workload functions as a whole when faults occur.
- Never experimenting under real-world conditions and expected load.
- Not treating your experiments as code or maintaining them through the development cycle.
- Not running chaos experiments both as part of your CI/CD pipeline, as well as outside of deployments.
- Neglecting to use past post-incident analyses when determining which faults to experiment with.

Benefits of establishing this best practice: Injecting faults to verify the resilience of your workload allows you to gain confidence that the recovery procedures of your resilient design will work in the case of a real fault.

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

Implementation guidance

Chaos engineering provides your teams with capabilities to continually inject real world disruptions (simulations) in a controlled way at the service provider, infrastructure, workload, and component level, with minimal to no impact to your customers. It allows your teams to learn from faults and observe, measure, and improve the resilience of your workloads, as well as validate that alerts fire and teams get notified in the case of an event.

When performed continually, chaos engineering can highlight deficiencies in your workloads that, if left unaddressed, could negatively affect availability and operation.

> **Note**
>
> 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

If a system is able to withstand these disruptions, the chaos experiment should be maintained as an automated regression test. In this way, chaos experiments should be performed as part of your systems development lifecycle (SDLC) and as part of your CI/CD pipeline.

To ensure that your workload can survive component failure, inject real world events as part of your experiments. For example, experiment with the loss of Amazon EC2 instances or failover of the primary Amazon RDS database instance, and verify that your workload is not impacted (or only minimally impacted). Use a combination of component faults to simulate events that may be caused by a disruption in an Availability Zone.

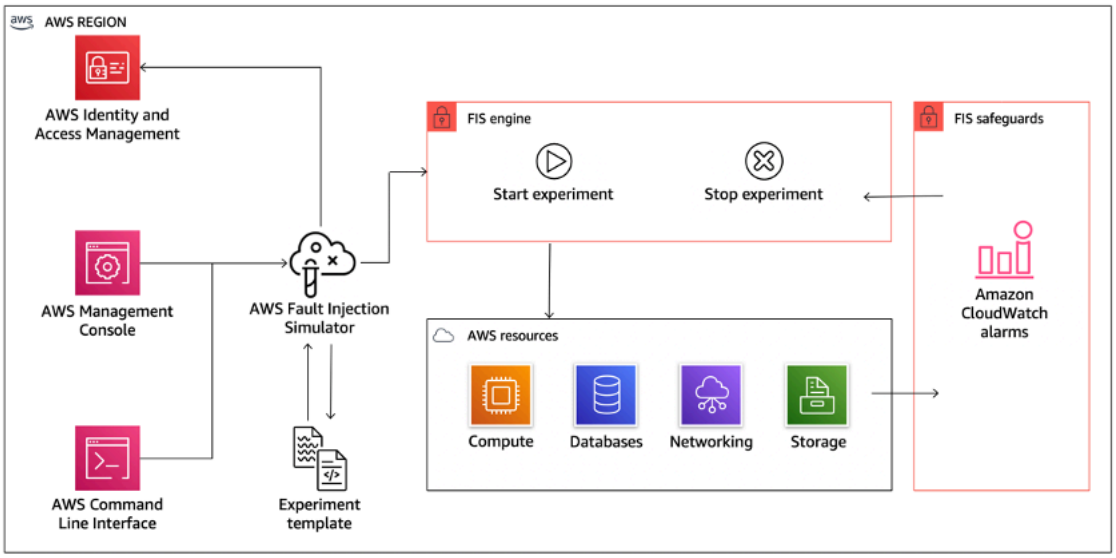
For application-level faults (such as crashes), you can start with stressors such as memory and CPU exhaustion.

To validate fallback or failover mechanisms for external dependencies due to intermittent network disruptions, your components should simulate such an event by blocking access to the third-party providers for a specified duration that can last from seconds to hours.

Other modes of degradation might cause reduced functionality and slow responses, often resulting in a disruption of your services. Common sources of this degradation are increased latency on critical services and unreliable network communication (dropped packets). Experiments with these faults, including networking effects such as latency, dropped messages, and DNS failures, could include the inability to resolve a name, reach the DNS service, or establish connections to dependent services.

Chaos engineering tools:

AWS Fault Injection Service (AWS FIS) is a fully managed service for running fault injection experiments that can be used as part of your CD pipeline, or outside of the pipeline. AWS FIS is a good choice to use during chaos engineering game days. It supports simultaneously introducing faults across different types of resources including Amazon EC2, Amazon Elastic Container Service (Amazon ECS), Amazon Elastic Kubernetes Service (Amazon EKS), and Amazon RDS. These faults include termination of resources, forcing failovers, stressing CPU or memory, throttling, latency, and packet loss. Since it is integrated with Amazon CloudWatch Alarms, you can set up stop conditions as guardrails to rollback an experiment if it causes unexpected impact.



AWS Fault Injection Service integrates with AWS resources to allow you to run fault injection experiments for your workloads.

There are also several third-party options for fault injection experiments. These include open-source tools such as Chaos Toolkit, Chaos Mesh, and Litmus Chaos, as well as commercial options like Gremlin. To expand the scope of faults that can be injected on AWS, AWS FIS integrates with Chaos Mesh and Litmus Chaos, allowing you to coordinate fault injection workflows among multiple tools. For example, you can run a stress test on a pod's CPU using Chaos Mesh or Litmus

faults while terminating a randomly selected percentage of cluster nodes using AWS FIS fault actions.

Implementation steps

1. Determine which faults to use for experiments.

   Assess the design of your workload for resiliency. Such designs (created using the best practices of the Well-Architected Framework) account for risks based on critical dependencies, past events, known issues, and compliance requirements. List each element of the design intended to maintain resilience and the faults it is designed to mitigate. For more information about creating such lists, see the Operational Readiness Review whitepaper which guides you on how to create a process to prevent reoccurrence of previous incidents. The Failure Modes and Effects Analysis (FMEA) process provides you with a framework for performing a component-level analysis of failures and how they impact your workload. FMEA is outlined in more detail by Adrian Cockcroft in Failure Modes and Continuous Resilience.

2. Assign a priority to each fault.

   Start with a coarse categorization such as high, medium, or low. To assess priority, consider frequency of the fault and impact of failure to the overall workload.

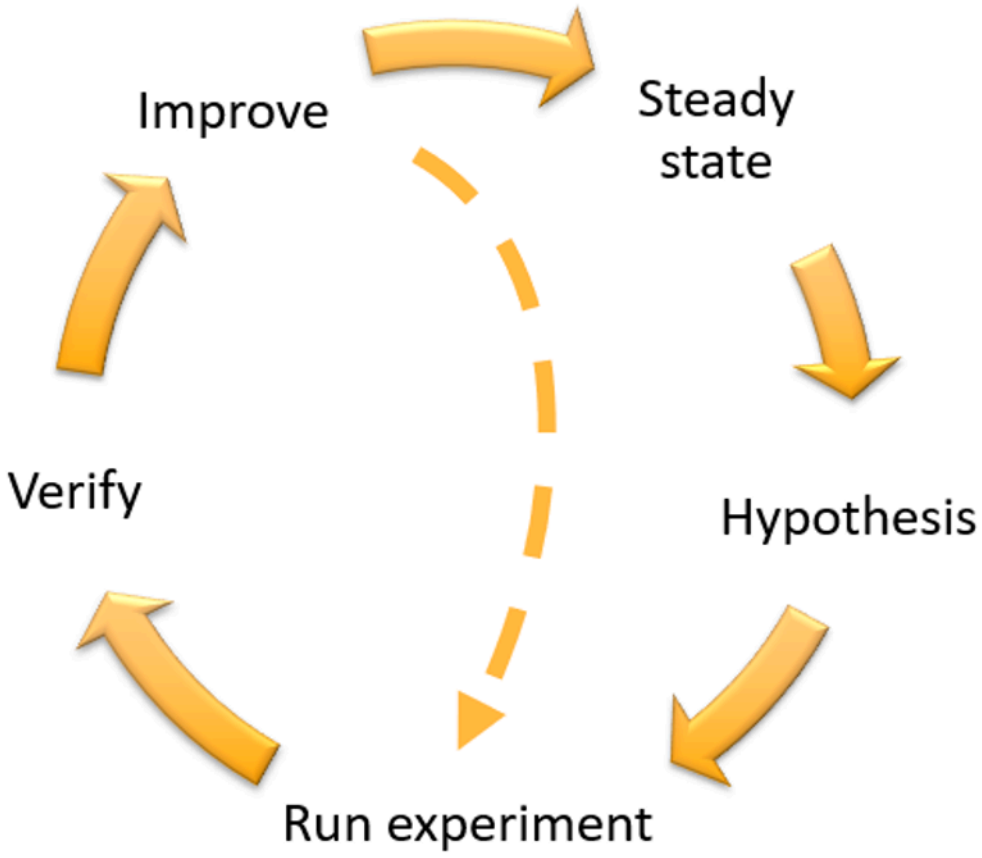
   When considering frequency of a given fault, analyze past data for this workload when available. If not available, use data from other workloads running in a similar environment.

   When considering impact of a given fault, the larger the scope of the fault, generally the larger the impact. Also consider the workload design and purpose. For example, the ability to access the source data stores is critical for a workload doing data transformation and analysis. In this case, you would prioritize experiments for access faults, as well as throttled access and latency insertion.

   Post-incident analyses are a good source of data to understand both frequency and impact of failure modes.

   Use the assigned priority to determine which faults to experiment with first and the order with which to develop new fault injection experiments.

3. For each experiment that you perform, follow the chaos engineering and continuous resilience flywheel in the following figure.



Chaos engineering and continuous resilience flywheel, using the scientific method by Adrian Hornsby.

a. Define steady state as some measurable output of a workload that indicates normal behavior.

   Your workload exhibits steady state if it is operating reliably and as expected. Therefore, validate that your workload is healthy before defining steady state. Steady state does not necessarily mean no impact to the workload when a fault occurs, as a certain percentage in faults could be within acceptable limits. The steady state is your baseline that you will observe during the experiment, which will highlight anomalies if your hypothesis defined in the next step does not turn out as expected.

   For example, a steady state of a payments system can be defined as the processing of 300 TPS with a success rate of 99% and round-trip time of 500 ms.

b. Form a hypothesis about how the workload will react to the fault.

A good hypothesis is based on how the workload is expected to mitigate the fault to maintain the steady state. The hypothesis states that given the fault of a specific type, the system or workload will continue steady state, because the workload was designed with specific mitigations. The specific type of fault and mitigations should be specified in the hypothesis.

The following template can be used for the hypothesis (but other wording is also acceptable):

> **Note**
>
> If *specific fault* occurs, the *workload name* workload will *describe mitigating controls* to maintain *business or technical metric impact*.

For example:

- If 20% of the nodes in the Amazon EKS node-group are taken down, the Transaction Create API continues to serve the 99th percentile of requests in under 100 ms (steady state). The Amazon EKS nodes will recover within five minutes, and pods will get scheduled and process traffic within eight minutes after the initiation of the experiment. Alerts will fire within three minutes.
- If a single Amazon EC2 instance failure occurs, the order system's Elastic Load Balancing health check will cause the Elastic Load Balancing to only send requests to the remaining healthy instances while the Amazon EC2 Auto Scaling replaces the failed instance, maintaining a less than 0.01% increase in server-side (5xx) errors (steady state).
- If the primary Amazon RDS database instance fails, the Supply Chain data collection workload will failover and connect to the standby Amazon RDS database instance to maintain less than 1 minute of database read or write errors (steady state).

c. Run the experiment by injecting the fault.

An experiment should by default be fail-safe and tolerated by the workload. If you know that the workload will fail, do not run the experiment. Chaos engineering should be used to find known-unknowns or unknown-unknowns. *Known-unknowns* are things you are aware of but don't fully understand, and *unknown-unknowns* are things you are neither aware of nor fully understand. Experimenting against a workload that you know is broken won't provide you with new insights. Your experiment should be carefully planned, have a clear scope of impact, and provide a rollback mechanism that can be applied in case of unexpected turbulence. If your due-diligence shows that your workload should survive the experiment, move forward

with the experiment. There are several options for injecting the faults. For workloads on AWS, AWS FIS provides many predefined fault simulations called actions. You can also define custom actions that run in AWS FIS using AWS Systems Manager documents.

We discourage the use of custom scripts for chaos experiments, unless the scripts have the capabilities to understand the current state of the workload, are able to emit logs, and provide mechanisms for rollbacks and stop conditions where possible.

An effective framework or toolset which supports chaos engineering should track the current state of an experiment, emit logs, and provide rollback mechanisms to support the controlled running of an experiment. Start with an established service like AWS FIS that allows you to perform experiments with a clearly defined scope and safety mechanisms that rollback the experiment if the experiment introduces unexpected turbulence. To learn about a wider variety of experiments using AWS FIS, also see the Resilient and Well-Architected Apps with Chaos Engineering lab. Also, AWS Resilience Hub will analyze your workload and create experiments that you can choose to implement and run in AWS FIS.

> **Note**
>
> For every experiment, clearly understand the scope and its impact. We recommend that faults should be simulated first on a non-production environment before being run in production.

Experiments should run in production under real-world load using canary deployments that spin up both a control and experimental system deployment, where feasible. Running experiments during off-peak times is a good practice to mitigate potential impact when first experimenting in production. Also, if using actual customer traffic poses too much risk, you can run experiments using synthetic traffic on production infrastructure against the control and experimental deployments. When using production is not possible, run experiments in pre-production environments that are as close to production as possible.

You must establish and monitor guardrails to ensure the experiment does not impact production traffic or other systems beyond acceptable limits. Establish stop conditions to stop an experiment if it reaches a threshold on a guardrail metric that you define. This should include the metrics for steady state for the workload, as well as the metric against the components into which you're injecting the fault. A synthetic monitor (also known as a user canary) is one metric you should usually include as a user proxy. Stop conditions for AWS FIS

are supported as part of the experiment template, allowing up to five stop-conditions per template.

One of the principles of chaos is minimize the scope of the experiment and its impact:

While there must be an allowance for some short-term negative impact, it is the responsibility and obligation of the Chaos Engineer to ensure the fallout from experiments are minimized and contained.

A method to verify the scope and potential impact is to perform the experiment in a non-production environment first, verifying that thresholds for stop conditions activate as expected during an experiment and observability is in place to catch an exception, instead of directly experimenting in production.

When running fault injection experiments, verify that all responsible parties are well-informed. Communicate with appropriate teams such as the operations teams, service reliability teams, and customer support to let them know when experiments will be run and what to expect. Give these teams communication tools to inform those running the experiment if they see any adverse effects.

You must restore the workload and its underlying systems back to the original known-good state. Often, the resilient design of the workload will self-heal. But some fault designs or failed experiments can leave your workload in an unexpected failed state. By the end of the experiment, you must be aware of this and restore the workload and systems. With AWS FIS you can set a rollback configuration (also called a post action) within the action parameters. A post action returns the target to the state that it was in before the action was run. Whether automated (such as using AWS FIS) or manual, these post actions should be part of a playbook that describes how to detect and handle failures.

d. Verify the hypothesis.

Principles of Chaos Engineering gives this guidance on how to verify steady state of your workload:

Focus on the measurable output of a system, rather than internal attributes of the system. Measurements of that output over a short period of time constitute a proxy for the system's steady state. The overall system's throughput, error rates, and latency percentiles could all be metrics of interest representing steady state behavior. By focusing on systemic behavior patterns during experiments, chaos engineering verifies that the system does work, rather than trying to validate how it works.

In our two previous examples, we include the steady state metrics of less than 0.01% increase in server-side (5xx) errors and less than one minute of database read and write errors.

The 5xx errors are a good metric because they are a consequence of the failure mode that a client of the workload will experience directly. The database errors measurement is good as a direct consequence of the fault, but should also be supplemented with a client impact measurement such as failed customer requests or errors surfaced to the client. Additionally, include a synthetic monitor (also known as a user canary) on any APIs or URIs directly accessed by the client of your workload.

e. Improve the workload design for resilience.

If steady state was not maintained, then investigate how the workload design can be improved to mitigate the fault, applying the best practices of the AWS Well-Architected Reliability pillar. Additional guidance and resources can be found in the AWS Builder's Library, which hosts articles about how to improve your health checks or employ retries with backoff in your application code, among others.

After these changes have been implemented, run the experiment again (shown by the dotted line in the chaos engineering flywheel) to determine their effectiveness. If the verify step indicates the hypothesis holds true, then the workload will be in steady state, and the cycle continues.

4. Run experiments regularly.

A chaos experiment is a cycle, and experiments should be run regularly as part of chaos engineering. After a workload meets the experiment's hypothesis, the experiment should be automated to run continually as a regression part of your CI/CD pipeline. To learn how to do this, see this blog on how to run AWS FIS experiments using AWS CodePipeline. This lab on recurrent AWS FIS experiments in a CI/CD pipeline allows you to work hands-on.

Fault injection experiments are also a part of game days (see REL12-BP05 Conduct game days regularly). Game days simulate a failure or event to verify systems, processes, and team responses. The purpose is to actually perform the actions the team would perform as if an exceptional event happened.

5. Capture and store experiment results.

Results for fault injection experiments must be captured and persisted. Include all necessary data (such as time, workload, and conditions) to be able to later analyze experiment results and

trends. Examples of results might include screenshots of dashboards, CSV dumps from your metric's database, or a hand-typed record of events and observations from the experiment. Experiment logging with AWS FIS can be part of this data capture.

Resources

Related best practices:

- REL08-BP03 Integrate resiliency testing as part of your deployment
- REL13-BP03 Test disaster recovery implementation to validate the implementation

Related documents:

- What is AWS Fault Injection Service?
- What is AWS Resilience Hub?
- Principles of Chaos Engineering
- Chaos Engineering: Planning your first experiment
- Resilience Engineering: Learning to Embrace Failure
- Chaos Engineering stories
- Avoiding fallback in distributed systems
- Canary Deployment for Chaos Experiments

Related videos:

- AWS re:Invent 2020: Testing resiliency using chaos engineering (ARC316)
- AWS re:Invent 2019: Improving resiliency with chaos engineering (DOP309-R1)
- AWS re:Invent 2019: Performing chaos engineering in a serverless world (CMY301)

Related tools:

- AWS Fault Injection Service
- AWS Marketplace: Gremlin Chaos Engineering Platform
- Chaos Toolkit

- Chaos Mesh
- Litmus

REL12-BP05 Conduct game days regularly

Conduct game days to regularly exercise your procedures for responding to workload-impacting events and impairments. Involve the same teams who would be responsible for handling production scenarios. These exercises help enforce measures to prevent user impact caused by production events. When you practice your response procedures in realistic conditions, you can identify and address any gaps or weaknesses before a real event occurs.

Game days simulate events in production-like environments to test systems, processes, and team responses. The purpose is to perform the same actions the team would perform as if the event actually occurred. These exercises help you understand where improvements can be made and can help develop organizational experience in dealing with events and impairments. These should be conducted regularly so that your team knows builds ingrained habits for how to respond.

Game days prepare teams to handle production events with greater confidence. Teams that are well-practiced are more able to quickly detect and respond to various scenarios. This results in a significantly improved readiness and resilience posture.

Desired outcome: You run resilience game days on a consistent, scheduled basis. These game days are seen as a normal and expected part of doing business. Your organization has built a culture of preparedness, and when production issues occur, your teams are well-prepared to respond effectively, resolve the issues efficiently, and mitigate the impact on customers.

Common anti-patterns:

- You document your procedures, but your never exercise them.
- You exclude business decision makers in the test exercises.
- You run a game day, but you don't inform all relevant stakeholders.
- You focus solely on technical failures, but you don't involve business stakeholders.
- You don't incorporate lessons learned from game days into your recovery processes.
- You blame teams for failures or bugs.

Benefits of establishing this best practice:

- Enhance response skills: On game days, teams practice their duties and test their communication mechanisms during simulated events, which creates a more coordinated and efficient response in production situations.
- Identify and address dependencies: Complex environments often involve intricate dependencies between various systems, services, and components. Game days can help you identify and address these dependencies, and verify that your critical systems and services are properly covered by your runbook procedures and can be scaled up or recovered in a timely manner.
- Foster a culture of resilience: Game days can help cultivate a mindset of resilience within an organization. When you involve cross-functional teams and stakeholders, these exercises promote awareness, collaboration, and a shared understanding of the importance of resilience across the entire organization.
- Continuous improvement and adaptation: Regular game days help you to continually assess and adapt your resilience strategies, which keeps them relevant and effective in the face of changing circumstances.
- Increase confidence in the system: Successful game days can help you build confidence in the system's ability to withstand and recover from disruptions.

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

Implementation guidance

Once you have designed and implemented the necessary resilience measures, conduct a game day to validate that everything works as planned in production. A game day, especially the first one, should involve all team members, and all stakeholders and participants should be informed in advance about the date, time, and simulated scenarios.

During the game day, the involved teams simulate various events and potential scenarios according to the prescribed procedures. The participants closely monitor and assess the impact of these simulated events. If the system operates as designed, the automated detection, scaling, and self-healing mechanisms should activate and result in little to no impact on users. If the team observes any negative impact, they roll back the test and remedy the identified issues, either through automated means or manual intervention documented in the applicable runbooks.

To continuously improve resilience, it's critical to document and incorporate lessons learned. This process is a *feedback loop* that systematically captures insights from game days and uses them to enhance systems, processes, and team capabilities.

To help you reproduce real-world scenarios where system components or services may fail unexpectedly, inject simulated faults as a game day exercise. Teams can test the resilience and fault tolerance of their systems and simulate their incident response and recovery processes in a controlled environment.

In AWS, your game days can be carried out with replicas of your production environment using infrastructure as code. Through this process, you can test in a safe environment that closely resembles your production environment. Consider AWS Fault Injection Service to create different failure scenarios. Use services like Amazon CloudWatch and AWS X-Ray to monitor system behavior during game days. Use AWS Systems Manager to manage and run playbooks, and use AWS Step Functions to orchestrate recurring game day workflows.

Implementation steps

- **Establish a game day program:** Develop a structured program that defines the frequency, scope and objectives of game days. Involve key stakeholders and subject matter experts in planning and running these exercises.
- **Prepare the game day:**
  1. Identify the key business-critical services that are the focus of the game day. Catalog and map the people, processes, and technologies that support those services.
  2. Set the agenda for the game day, and prepare the involved teams to participate in the event. Prepare your automation services to simulate the planned scenarios and run the appropriate recovery processes. AWS services such as AWS Fault Injection Service, AWS Step Functions, and AWS Systems Manager can help you automate various aspects of game days, such as injection of faults and initiation of recovery actions.
- **Run your simulation:** On the game day, run the planned scenario. Observe and document how the people, processes, and technologies react to the simulated event.
- **Conduct post-exercise reviews:** After the game day, conduct a retrospective session to review the lessons learned. Identify areas for improvement and any actions needed to improve operational resilience. Document your findings, and track any necessary changes to enhance your resilience strategies and preparedness to completion.

Resources

Related best practices:

- REL12-BP01 Use playbooks to investigate failures

- REL12-BP04 Test resiliency using chaos engineering
- OPS04-BP01 Identify key performance indicators
- OPS07-BP03 Use runbooks to perform procedures
- OPS10-BP01 Use a process for event, incident, and problem management

Related documents:

- What is AWS GameDay?
- AWS Well-Architected Concepts - Game Day

Related videos:

- AWS re:Invent 2023 - Practice like you play: How Amazon scales resilience to new heights

Related examples:

- AWS Workshop - Navigate the storm: Unleashing controlled chaos for resilient systems
- Build Your Own Game Day to Support Operational Resilience

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 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

- REL13-BP02 Use defined recovery strategies to meet the recovery objectives
- REL13-BP03 Test disaster recovery implementation to validate the implementation
- REL13-BP04 Manage configuration drift at the DR site or Region
- REL13-BP05 Automate recovery

REL13-BP01 Define recovery objectives for downtime and data loss

Failures can impact your business in several ways. First, failures can cause service interruption (downtime). Second, failures can cause data to become lost, inconsistent, or stale. In order to guide how you respond and recover from failures, define a Recovery Time Objective (RTO) and Recovery Point Objective (RPO) for each workload. *Recovery Time Objective (RTO)* is the maximum acceptable delay between the interruption of service and restoration of service. *Recovery Point Objective (RPO)* is the maximum acceptable time after the last data recovery point.

Desired outcome: Every workload has a designated RTO and RPO based on technical considerations and business impact.

Common anti-patterns:

- You haven't designated recovery objectives.
- You select arbitrary recovery objectives.
- You select recovery objectives that are too lenient and do not meet business objectives.
- You have not evaluated the impact of downtime and data loss.
- You select unrealistic recovery objectives, such as zero time to recover or zero data loss, which may not be achievable for your workload configuration.
- You select recovery objectives that are more stringent than actual business objectives. This forces recovery implementations that are costlier and more complicated than what the workload needs.
- You select recovery objectives that are incompatible with those of a dependent workload.
- You fail to consider regulatory and compliance requirements.

Benefits of establishing this best practice: When you set RTOs and RPOs for your workloads, you establish clear and measurable goals for recovery based on your business needs. Once you've set those goals, you can create disaster recovery (DR) plans that are tailored to meet them.

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

Implementation guidance

Construct a matrix or worksheet to help guide your disaster recovery planning. In your matrix, create different workload categories or tiers based on their business impact (such as critical, high, medium, and low) and the associated RTOs and RPOs to target for each one. The following matrix provides an example (note that your RTO and RPO values may differ) you can follow:

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

Example disaster recovery matrix

For each workload, investigate and understand the impact of downtime and lost data on your business. The impact typically grows with downtime and data loss, but the shape of the impact can differ based on the workload type. For example, downtime for up to an hour might have low impact, but after that, the impact could quickly intensify. Impact can take many forms, including financial impact (such as lost revenue), reputational impact (including loss of customer trust), operational impact (such as a missed payroll or decreased productivity), and regulatory risk. Once completed, assign the workload to the appropriate tier.

Consider the following questions when you analyze the impact of failure:

1. What is the maximum time the workload can be unavailable before unacceptable impact to the business is incurred?
2. How much impact, and what kind, will be incurred by the business by a workload disruption? Consider all kinds of impact, including financial, reputational, operational, and regulatory.
3. What is the maximum amount of data that can be lost or unrecoverable before unacceptable impact to the business is incurred?

4. Can lost data be recreated from other sources (also known as *derived* data)? If so, also consider the RPOs of all source data used to recreate the workload data.
5. What are the recovery objectives and availability expectations of workloads that this one depends on (downstream)? Your workload's objectives must be achievable given the recovery capabilities of its downstream dependencies. Consider possible downstream dependency workarounds or mitigations that can improve this workload's recovery capability.
6. What are the recovery objectives and availability expectations of workloads that depend on this one (upstream)? Upstream workload objectives may require this workload to have more stringent recovery capabilities than it first appears.
7. Are there different recovery objectives based on the type of incident? For example, you might have different RTOs and RPOs depending on whether the incident impacts an Availability Zone or an entire Region.
8. Do your recovery objectives change during certain events or times of the year? For example, you might have different RTOs and RPOs around holiday shopping seasons, sporting events, special sales, and new product launches.
9. How do the recovery objectives align with any line of business and organizational disaster recovery strategy you might have?
10. Are there legal or contractual ramifications to consider? For example, are you contractually obligated to provide a service with a given RTO or RPO? What penalties might you incur for not meeting them?
11. Are you required to maintain data integrity to meet regulatory or compliance requirements?

The following worksheet can aid your evaluation of each workload. You may modify 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

Implementation steps

1. Identify the business stakeholders and technical teams responsible for each workload, and engage with them.
2. Create categories or tiers of criticality for workload impact in your organization. Example categories include critical, high, medium, and low. For each category, choose an RTO and RPO that reflects your business objectives and requirements.
3. Assign one of the impact categories you created in the previous step to each workload. To decide how a workload maps to a category, consider the workload's importance to the business and the impact of interruption or data loss, and use the questions above to guide you. This results in an RTO and RPO for each workload.
4. Consider the RTO and RPO for each workload determined in the previous step. Involve the workload's business and technical teams to determine whether the objectives should be adjusted. For example, business stakeholders could determine that more stringent targets are required. Alternatively, technical teams could determine that targets should be modified to make them achievable with available resources and technological constraints.

Resources

Related best practices:

- REL09-BP04 Perform periodic recovery of the data to verify backup integrity and processes
- REL12-BP01 Use playbooks to investigate failures
- REL13-BP02 Use defined recovery strategies to meet the recovery objectives
- REL13-BP03 Test disaster recovery implementation to validate the implementation

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
- 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.

Desired outcome: For each workload, there is a defined and implemented DR strategy that allows the 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 disaster recovery.
- 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 improves knowledge sharing between teams and 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

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.

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.

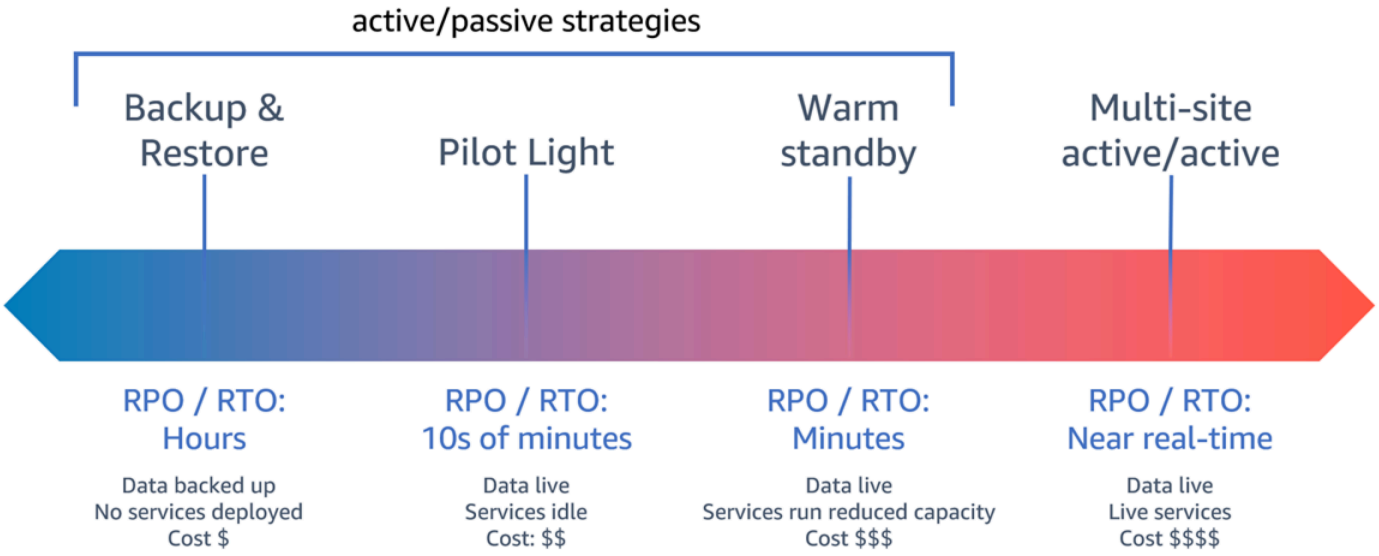


Figure 17: Disaster recovery (DR) strategies

- **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 permit point in time recovery (PITR), 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.
> When cost is a concern, and you wish to achieve a similar RPO and RTO objectives as defined in the warm standby strategy, you could consider cloud native solutions, like AWS Elastic Disaster Recovery, that take the pilot light approach and offer improved RPO and RTO targets.

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) and the 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.

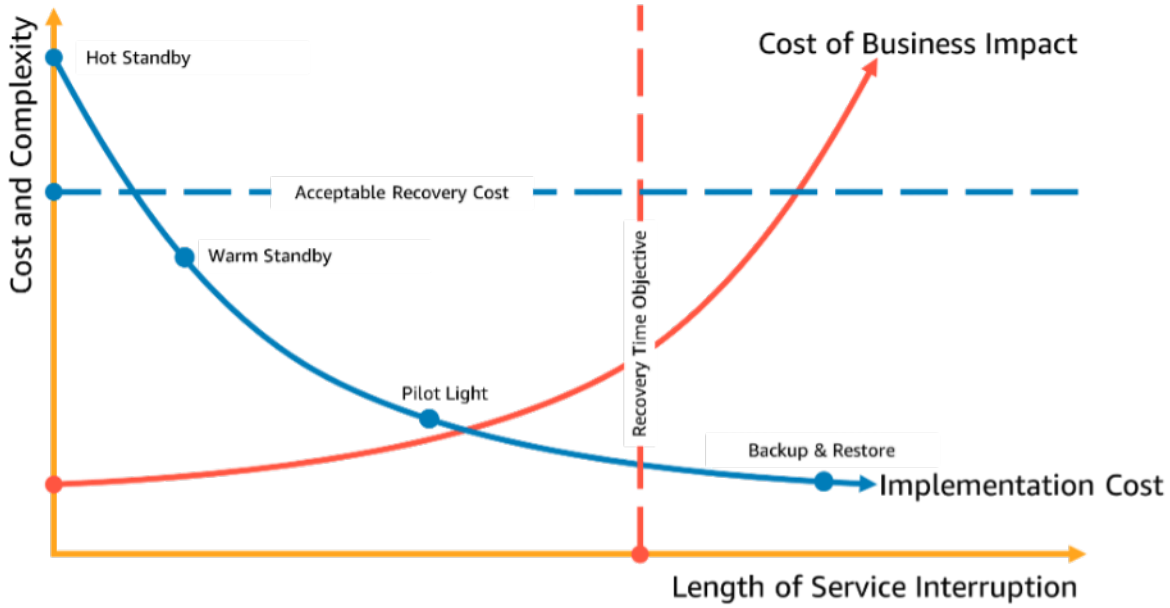


Figure 18: Choosing a DR strategy based on RTO and cost

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 following steps, you can 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).

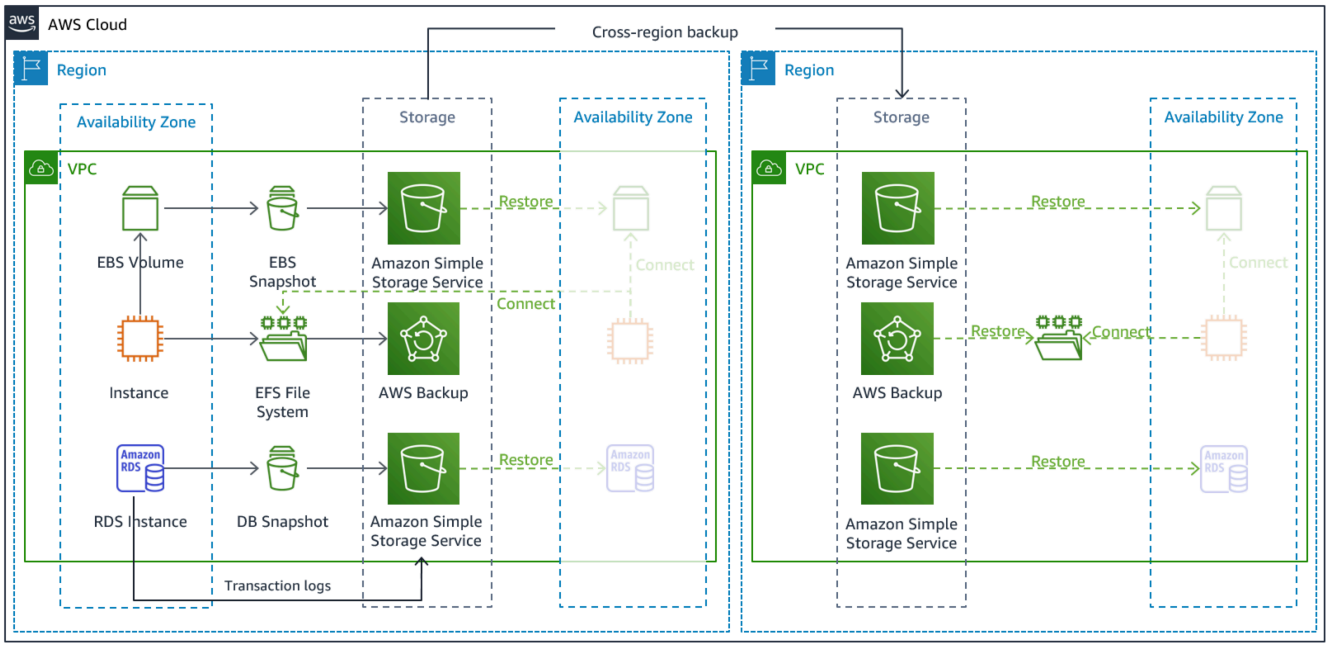


Figure 19: Backup and restore architecture

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 20, no compute instances are deployed.

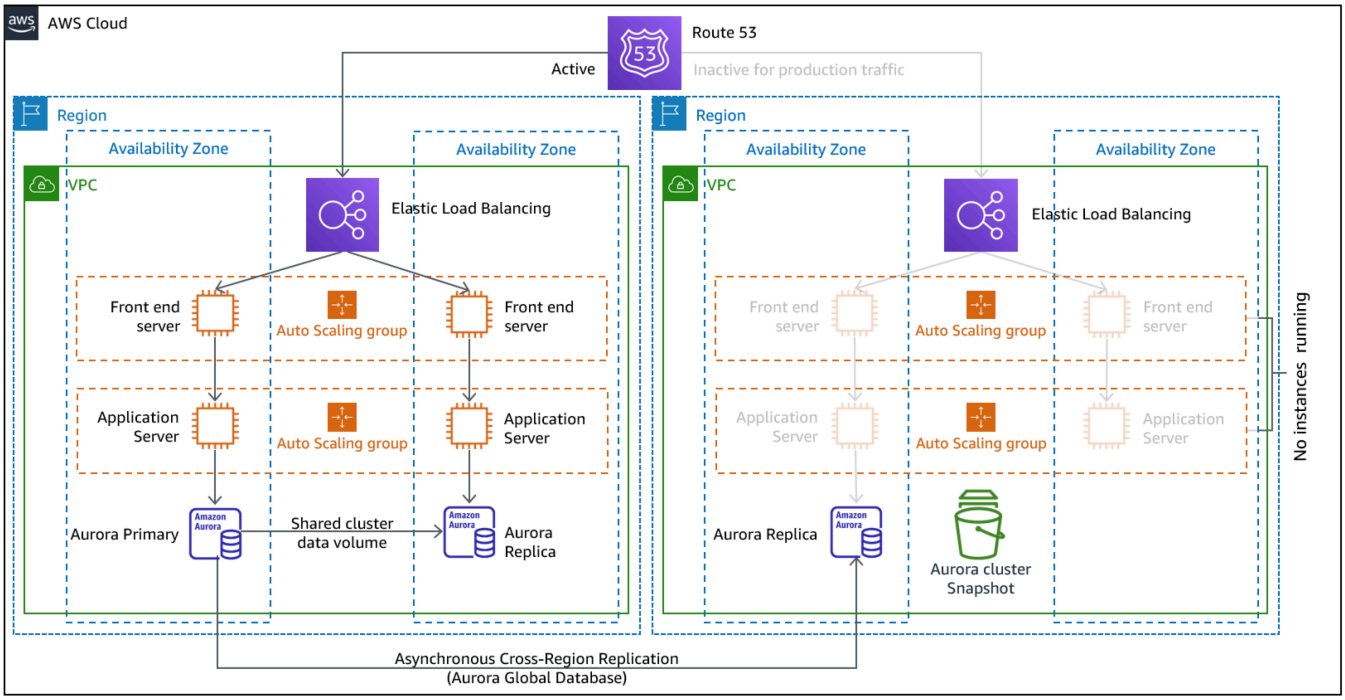


Figure 20: Pilot light architecture

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*.

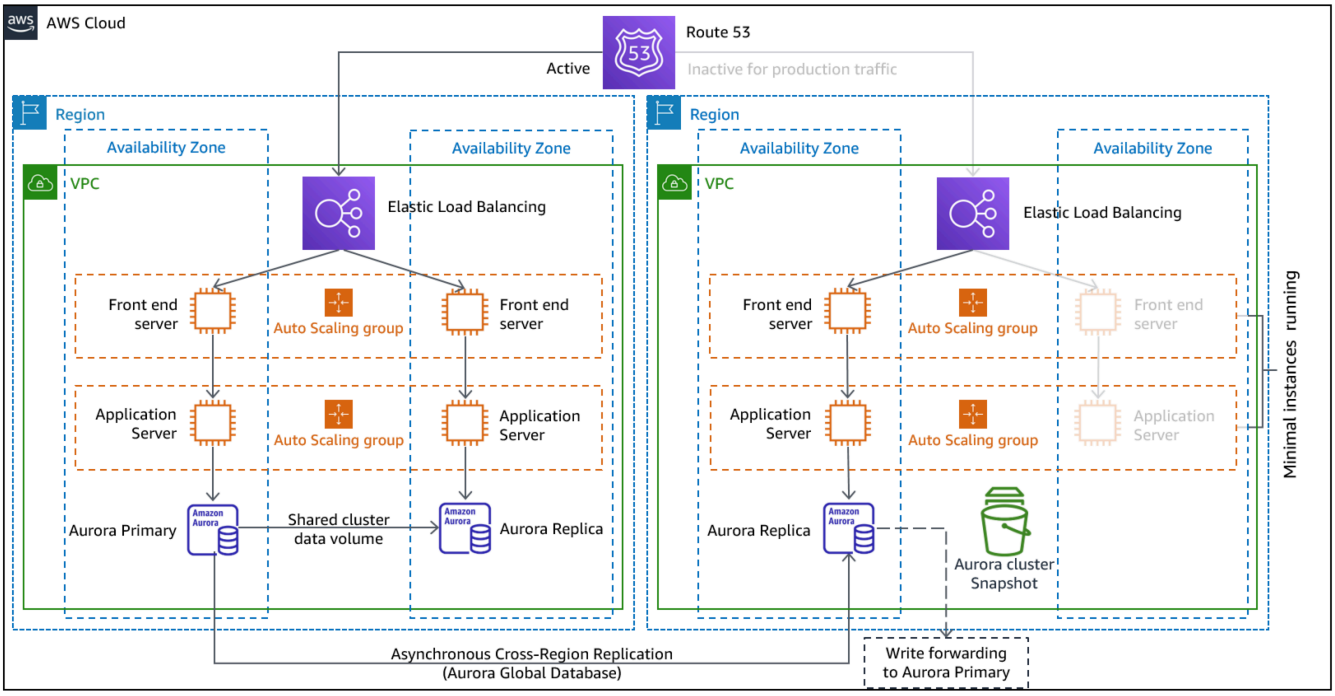


Figure 21: Warm standby architecture

Using warm standby or pilot light requires scaling up resources in the recovery Region. To verify capacity is available when needed, consider the use for capacity reservations for EC2 instances. If using AWS Lambda, then provisioned concurrency can provide runtime 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 Regions 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.

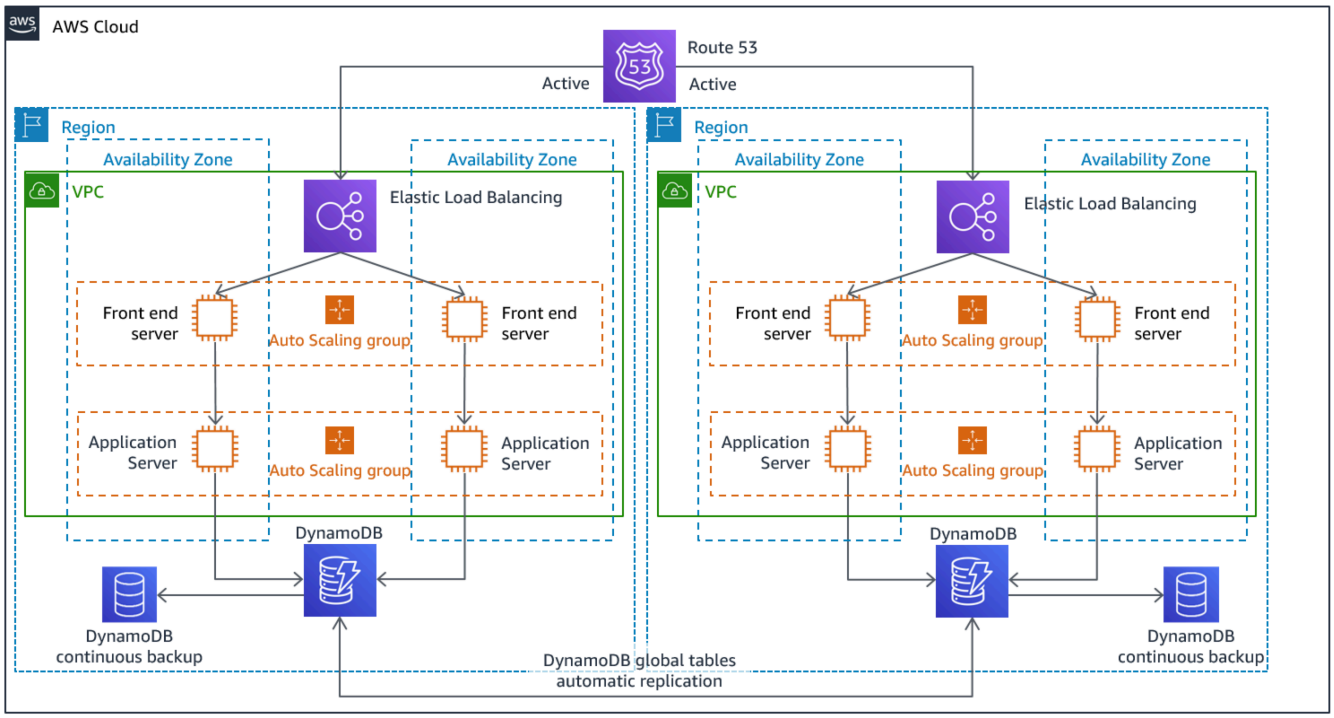


Figure 22: Multi-site active/active architecture

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

AWS Elastic Disaster Recovery

If you are considering the pilot light or warm standby strategy for disaster recovery, AWS Elastic Disaster Recovery could provide an alternative approach with improved benefits. Elastic Disaster Recovery can offer an RPO and RTO target similar to warm standby, but maintain the low-cost approach of pilot light. Elastic Disaster Recovery replicates your data from your primary region to your recovery Region, using continual data protection to achieve an RPO measured in seconds and an RTO that can be measured in minutes. Only the resources required to replicate the data are deployed in the recovery region, which keeps costs down, similar to the pilot light strategy. When using Elastic Disaster Recovery, the service coordinates and orchestrates the recovery of compute resources when initiated as part of failover or drill.

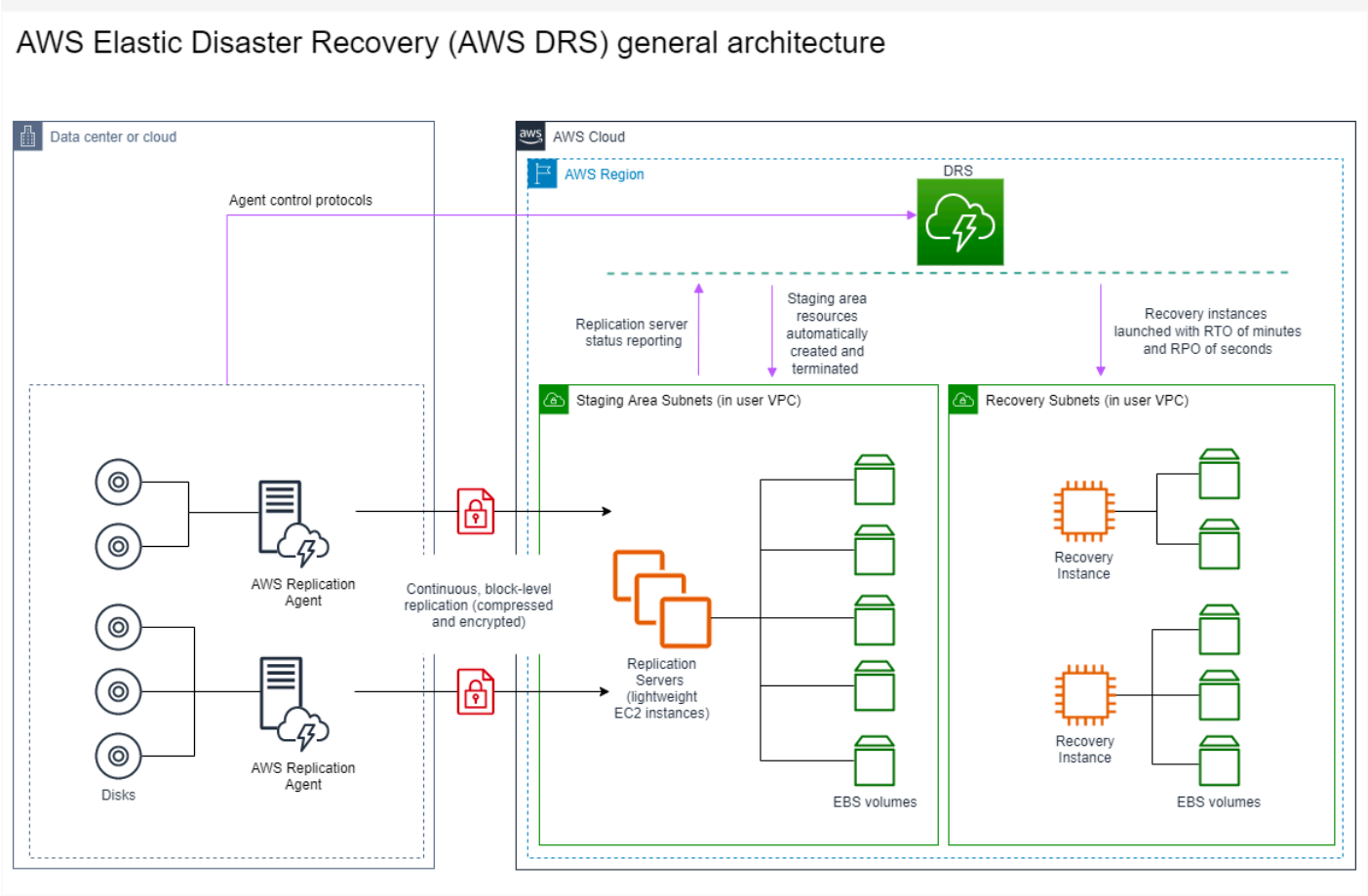


Figure 23: AWS Elastic Disaster Recovery architecture

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 23. 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 24: 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 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 use 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. When using Elastic Disaster Recovery, the service will replicate and orchestrate the restoration of application configurations and compute resources.

   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 20, 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 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 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.

   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.

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.

When using Elastic Disaster Recovery, the service will assist in orchestrating and automating the failback process. For more details, see Performing a failback.

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”
- the section called “REL11-BP04 Rely on the data plane and not the control plane during recovery”
- the section called “REL13-BP01 Define recovery objectives for downtime and data loss”

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 Amazon 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

REL13-BP03 Test disaster recovery implementation to validate the implementation

Regularly test failover to your recovery site to verify that it operates properly and that RTO and RPO are met.

Common anti-patterns:

- Never exercise failovers in production.

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

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

Implementation guidance

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.

Implementation steps

1. Engineer your workloads for recovery. Regularly test your recovery paths. Recovery-oriented computing identifies the characteristics in systems that enhance recovery: 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 verify 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.
2. For Amazon EC2-based workloads, use AWS Elastic Disaster Recovery to implement and launch drill instances for your DR strategy. AWS Elastic Disaster Recovery provides the ability to efficiently run drills, which helps you prepare for a failover event. You can also frequently launch of your instances using Elastic Disaster Recovery for test and drill purposes without redirecting the traffic.

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

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

To perform a successful disaster recovery (DR) procedure, your workload must be able to resume normal operations in a timely manner with no relevant loss of functionality or data once the DR environment has been brought online. To achieve this goal, it's essential to maintain consistent infrastructure, data, and configurations between your DR environment and the primary environment.

Desired outcome: Your disaster recovery site's configuration and data are in parity with the primary site, which facilitates rapid and complete recovery when needed.

Common anti-patterns:

- You fail to update recovery locations when changes are made to the primary locations, which results in outdated configurations that could hinder recovery efforts.
- You do not consider potential limitations such as service differences between primary and recovery locations, which can lead to unexpected failures during failover.
- You rely on manual processes to update and synchronize the DR environment, which increases the risk of human error and inconsistency.
- You fail to detect configuration drift, which leads to a false sense of DR site readiness prior to an incident.

Benefits of establishing this best practice: Consistency between the DR environment and the primary environment significantly improves the likelihood of a successful recovery after an incident and reduces the risk of a failed recovery procedure.

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

Implementation guidance

A comprehensive approach to configuration management and failover readiness can help you verify that the DR site is consistently updated and prepared to take over in the event of a primary site failure.

To achieve consistency between your primary and disaster recovery (DR) environments, validate that your delivery pipelines distribute applications to both your primary and DR sites. Roll out changes to the DR sites after an appropriate evaluation period (also known as *staggered deployments*) to detect problems at the primary site and halt the deployment before they spread. Implement monitoring to detect configuration drift, and track changes and compliance across your environments. Perform automated remediation in the DR site to keep it fully consistent and ready to take over in the event of an incident.

Implementation steps

1. Validate that the DR region contains the AWS services and features required for a successful execution of your DR plan.
2. Use infrastructure as code (IaC). Keep your production infrastructure and application configuration templates accurate, and regularly apply them to your disaster recovery environment. AWS CloudFormation can detect drift between what your CloudFormation templates specify and what is actually deployed.
3. Configure CI/CD pipelines to deploy applications and infrastructure updates to all environments, including primary and DR sites. CI/CD solutions such as AWS CodePipeline can automate the deployment process, which reduces the risk of configuration drift.
4. Stagger deployments between the primary and DR environments. This approach allows updates to be initially deployed and tested in the primary environment, which isolates issues in the primary site before they are propagated to the DR site. This approach prevents defects from being simultaneously pushed to production and the DR site at the same time and maintains the integrity of the DR environment.
5. Continually monitor resource configurations in both primary and DR environments. Solutions such as AWS Config can help to enforce configuration compliance and detect drift, which helps maintain the consistent configurations across environments.
6. Implement alerting mechanisms to track and notify upon any configuration drift or data replication interruption or lag.
7. Automate the remediation of detected configuration drift.

8. Schedule regular audits and compliance checks to verify ongoing alignment between primary and DR configurations. Periodic reviews help you maintain compliance with defined rules and identify any discrepancies that need to be addressed.
9. Check for mismatches in AWS provisioned capacity, service quotas, throttle limits, and configuration and version discrepancies.

Resources

Related best practices:

- REL01-BP01 Aware of service quotas and constraints
- REL01-BP02 Manage service quotas across accounts and regions
- REL01-BP04 Monitor and manage quotas
- REL13-BP03 Test disaster recovery implementation to validate the implementation

Related documents:

- Remediating Noncompliant AWS Resources by AWS Config Rules
- AWS Systems Manager Automation
- AWS CloudFormation: Detecting unmanaged configuration changes to stacks and resources
- AWS CloudFormation: Detect Drift on an Entire CloudFormation Stack
- 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)

Related examples:

- AWS CloudFormation Registry

- Quota Monitor for AWS
- Implement automatic drift remediation for AWS CloudFormation using Amazon CloudWatch and AWS Lambda
- AWS Architecture Blog: Disaster Recovery Series
- AWS Marketplace: products that can be used for disaster recovery
- Automating safe, hands-off deployments

REL13-BP05 Automate recovery

Implement tested and automated recovery mechanisms that are reliable, observable, and reproducible to reduce the risk and business impact of failure.

Desired outcome: You have implemented a well-documented, standardized, and thoroughly-tested automation workflow for recovery processes. Your recovery automation automatically corrects minor issues that pose low risk of data loss or unavailability. You are able to quickly invoke recovery processes for serious incidents, observe the remediation behavior while they operate, and end the processes if you observe dangerous situations or failures.

Common anti-patterns:

- You depend on components or mechanisms that are in a failed or degraded state as part of your recovery plan.
- Your recovery processes require manual intervention, such as console access (also known as *click ops*).
- You automatically initiate recovery procedures in situations that present a high risk of data loss or unavailability.
- You fail to include a mechanism to abort a recovery procedure (like an *Andon cord* or *big red stop button*) that is not working or that poses additional risks.

Benefits of establishing this best practice:

- Increased reliability, predictability, and consistency of recovery operations.
- Ability to meet more stringent recovery objectives, including Recovery Time Objective (RTO) and Recovery Point Objective (RPO).
- Reduced likelihood of recovery failing during an incident.

- Reduced risk of failures associated with manual recovery processes that are prone to human error.

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

Implementation guidance

To implement automated recovery, you need a comprehensive approach that uses AWS services and best practices. To start, identify critical components and potential failure points in your workload. Develop automated processes that can recover your workloads and data from failures without human intervention.

Develop your recovery automation using infrastructure as code (IaC) principles. This makes your recovery environment consistent with the source environment and allows for version control of your recovery processes. To orchestrate complex recovery workflows, consider solutions such as AWS Systems Manager Automations or AWS Step Functions.

Automation of recovery processes provides significant benefits and can help you more easily achieve your Recovery Time Objective (RTO) and Recovery Point Objective (RPO). However, they can encounter unexpected situations that may cause them to fail or create new risks of their own such as additional downtime and data loss. To mitigate this risk, provide the ability to quickly halt a recovery automation in progress. Once halted, you can investigate and take corrective steps.

For supported workloads, consider solutions such as AWS Elastic Disaster Recovery (AWS DRS) to provide automated failover. AWS DRS continually replicates your machines (including operating system, system state configuration, databases, applications, and files) into a staging area in your target AWS account and preferred Region. If an incident occurs, AWS DRS automates the conversion of your replicated servers into fully-provisioned workloads in your recovery Region on AWS.

Maintenance and improvement of automated recovery is an ongoing process. Continually test and refine your recovery procedures based on lessons learned, and stay updated on new AWS services and features that can enhance your recovery capabilities.

Implementation steps

1. **Plan for automated recovery**
    a. Conduct a thorough review of your workload architecture, components, and dependencies to identify and plan automated recovery mechanisms. Categorize your workload's dependencies

into *hard* and *soft* dependencies. Hard dependencies are those that the workload cannot operate without and for which no substitute can be provided. Soft dependencies are those that the workload ordinarily uses but are replaceable with temporary substitute systems or processes or can be handled by graceful degradation.

b. Establish processes to identify and recover missing or corrupted data.

c. Define steps to confirm a recovered steady state after recovery actions have been completed.

d. Consider any actions required to make the recovered system ready for full service, such as pre-warming and populating caches.

e. Consider problems that could be encountered during the recovery process and how to detect and remediate them.

f. Consider scenarios where the primary site and its control plane are inaccessible. Verify that recovery actions can be performed independently without reliance on the primary site. Consider solutions such as Amazon Application Recovery Controller (ARC) to redirect traffic without the need to manually mutate DNS records.

2. **Develop automated recovery process**

a. Implement automated fault detection and failover mechanisms for hands-free recovery. Build dashboards such as with Amazon CloudWatch to report the progress and health of automated recovery procedures. Include procedures to validate successful recovery. Provide a mechanism to abort a recovery in process.

b. Build playbooks as a fallback process for faults that cannot be automatically recovered from, and take into consideration your disaster recovery plan.

c. Test recovery processes as discussed in REL13-BP03.

3. **Prepare for recovery**

a. Evaluate the state of your recovery site and deploy critical components to it in advance. For more detail, see REL13-BP04.

b. Define clear roles, responsibilities, and decision-making processes for recovery operations, involving relevant stakeholders and teams across the organization.

c. Define the conditions to initiate your recovery processes.

d. Create a plan to revert the recovery process and fall back to your primary site if required or after it's considered safe.

Resources

Related best practices:

- REL07-BP01 Use automation when obtaining or scaling resources
- REL11-BP01 Monitor all components of the workload to detect failures
- REL13-BP02 Use defined recovery strategies to meet the recovery objectives
- REL13-BP03 Test disaster recovery implementation to validate the implementation
- REL13-BP04 Manage configuration drift at the DR site or Region

Related documents:

- AWS Architecture Blog: Disaster Recovery Series
- Disaster Recovery of Workloads on AWS: Recovery in the Cloud (AWS Whitepaper)
- Orchestrate Disaster Recovery Automation using Amazon Route 53 ARC and AWS Step Functions
- Build AWS Systems Manager Automation runbooks using AWS CDK
- AWS Marketplace: Products That Can Be Used for Disaster Recovery
- AWS Systems Manager Automation
- AWS Elastic Disaster Recovery
- Using Elastic Disaster Recovery for Failover and Failback
- AWS Elastic Disaster Recovery Resources
- APN Partner: Partners That Can Help with Disaster Recovery

Related videos:

- AWS re:Invent 2018: Architecture Patterns for Multi-Region Active-Active Applications (ARC209-R2)
- AWS re:Invent 2022: AWS On Air ft. AWS Failback for AWS Elastic Disaster Recovery

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 challenged 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:

- Michael Fischer, Principal Solutions Architect, Amazon Web Services
- Seth Eliot, Principal Developer Advocate, 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, 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
- Laurent Domb, Chief Technologist - Fed Fin, Amazon Web Services
- Kevin Schwarz, Sr. Solutions Architect, Amazon Web Services
- Rob Martell, Principal Cloud Resilience Architect, Amazon Web Services
- Priyam Reddy, Senior Solutions Architect Manager DR, Amazon Web Services
- Jeff Ferris, Principal Technologist, Amazon Web Services
- Matias Battaglia, Senior Solutions Architect, 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
Updated best practice guidance	Best practices were updated with new guidance in the following areas: REL 1, REL 2, REL 4, REL 6, REL 7, REL 8, REL 10, REL 12, and REL 13. Guidance has been expanded and clarified throughout the pillar. REL10-BP02 and REL12-BP03 have had their guidance merged into other best practices. Resources throughout the pillar have been updated.	November 6, 2024
Updated best practice guidance	Small updates to best practices in REL 2, 4, 5, 6, 7, and 8.	June 27, 2024
Updated best practice guidance	Best practices were updated with new guidance in the following areas: Design interactions in a distributed system to prevent failures, Design interactions in a distributed system to mitigate or withstand failures, Monitor workload resources, Design your workload to adapt to changes in demand,	December 6, 2023

	Implement change, and Test reliability.	
Updated best practice guidance	Best practices were updated with new guidance in the following areas: Monitor workload resources and Design your workload to withstand component failures.	October 3, 2023
Updated best practice guidance	Best practices were updated with new guidance in the following areas: Design your workload service architecture, Design interactions in a distributed system to mitigate or withstand failures, and Monitor workload resources.	July 13, 2023
Minor update	Remove non-inclusive language.	April 13, 2023
Updates for new Framework	Best practices updated with prescriptive guidance and new best practices added.	April 10, 2023
Whitepaper updated	Best practices updated with new implementation guidance.	December 15, 2022
Minor updates	Corrected figure numbers and minor changes throughout.	November 17, 2022
Whitepaper updated	Best practices expanded and improvement plans added.	October 20, 2022

Whitepaper updated	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
Whitepaper updated	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	Added information about AWS Fault Injection Service (AWS FIS).	March 15, 2021
Minor update	Minor text update.	January 4, 2021

Whitepaper updated	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 reference d book to latest version; Expand availability calculati ons to include request-based calculation, and shortcut calculations; Improve descripti on for Game Days	December 7, 2020
Minor update	Updated Appendix A to update the Availability Design Goal for AWS Lambda	October 27, 2020
Minor update	Updated Appendix A to add the Availability Design Goal for AWS Global Accelerator	July 24, 2020

Updates for new Framework	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	Fixed broken link	October 1, 2019
Whitepaper updated	Appendix A updated	April 1, 2019

Whitepaper updated	Added specific AWS Direct Connect networking recommendations and additional service design goals	September 1, 2018
Whitepaper updated	Added Design Principles and Limit Management sections. Updated links, removed ambiguity of upstream/ downstream terminology, and added explicit reference s to the remaining Reliability Pillar topics in the availability scenarios.	June 1, 2018
Whitepaper updated	Changed DynamoDB Cross Region solution to DynamoDB Global Tables. Added service design goals	March 1, 2018
Minor updates	Minor correction to availabil ity calculation to include application availability	December 1, 2017
Whitepaper updated	Updated to provide guidance on high availability designs, including concepts, best practice and example implementations.	November 1, 2017
Initial publication	Reliability Pillar - AWS Well-Architected Framework published.	November 1, 2016

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.