Architecting for the Cloud

AWS Best Practices

October 2018

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. 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.

Contents

Abstract

This whitepaper is intended for solutions architects and developers who are building solutions that will be deployed on Amazon Web Services (AWS). It provides architectural guidance and advice on technical design patterns and how they are applied in the context of cloud computing. This introduction provides the key concepts and differences when designing solutions on AWS. It includes a discussion on how to take advantage of attributes that are specific to the dynamic nature of cloud computing, such as elasticity and infrastructure automation. These patterns can provide the context for a more detailed review of choices, operational status, and implementation status as detailed in the *AWS Well-Architected Framework*.[1]

Introduction

Migrating applications to AWS, even without significant changes (an approach known as *lift and shift*), provides organizations with the benefits of a secure and cost-efficient infrastructure. However, to make the most of the elasticity and agility that are possible with cloud computing, engineers have to evolve their architectures to take advantage of AWS capabilities.

For new applications, cloud-specific IT architecture patterns can help drive efficiency and scalability. Those new architectures can support anything from real-time analytics of internet-scale data to applications with unpredictable traffic from thousands of connected Internet of Things (IoT) or mobile devices.

Whether you are rearchitecting the applications that currently run in your on-premises environment to run on AWS, or designing cloud-native applications, you must consider the differences between traditional environments and cloud computing environments. This includes architecture choices, scalability, resource types, automation, as well as flexible components, services, and databases. If you are new to AWS, we recommend that you review the information on the About AWS page to get a basic understanding of AWS services.[2]

Differences Between Traditional and Cloud Computing Environments

Cloud computing differs from a traditional, on-premises environment in many ways, including flexible, global, and scalable capacity, managed services, built-in security, options for cost optimization, and various operating models.

IT Assets as Provisioned Resources

In a traditional computing environment, you provision capacity based on an estimate of a theoretical maximum peak. This can result in periods where expensive resources are sitting idle or occasions of insufficient capacity. With cloud computing, you can access as much or as little capacity as you need and dynamically scale to meet actual demand, while only paying for what you use.

On AWS, servers, databases, storage, and higher-level application components can be instantiated within seconds. You can treat these as temporary resources, free from the inflexibility and constraints of a fixed and finite IT infrastructure. This resets the way you approach change management, testing, reliability, and capacity planning. This change in approach encourages experimentation by introducing the ability in processes to fail fast and iterate quickly.

Global, Available, and Scalable Capacity

Using the global infrastructure of AWS, you can deploy your application to the AWS Region that best meets your requirements (e.g., proximity to your end users, compliance, data residency constraints, and cost).[3] For global applications, you can reduce latency to end users around the world by using the Amazon CloudFront content delivery network (CDN). This also makes it much easier to operate production applications and databases across multiple data centers to achieve high availability and fault tolerance. The global infrastructure of AWS and the ability to provision capacity as needed let you think differently about your infrastructure as the demands on your applications and the breadth of your services expand.

Higher-Level Managed Services

Apart from the compute resources of Amazon Elastic Compute Cloud (Amazon EC2), you also have access to a broad set of storage, database, analytics, application, and deployment services. Because these services are instantly available to developers, they reduce dependency on in-house specialized skills and allow organizations to deliver new solutions faster. AWS services that are managed can lower operational complexity and cost. They are also designed for scalability and high availability, so they can reduce risk for your implementations.

Built-in Security

In traditional IT environments, infrastructure security auditing can be a periodic and manual process. In contrast, the AWS Cloud provides governance capabilities that enable continuous monitoring of configuration changes to your IT resources. Security at AWS is the highest priority, which means that you benefit from data centers and network architecture that are built to meet the requirements of the most security-sensitive organizations.

Since AWS resources are programmable using tools and APIs, you can formalize and embed your security policy within the design of your infrastructure. With the ability to spin up temporary environments, security testing can now become part of your continuous delivery pipeline. Finally, you can leverage a variety of native AWS security and encryption features that can help you achieve higher levels of data protection and compliance.

Architecting for Cost

Traditional cost management of on-premises solutions is not typically tightly coupled to the provision of services. When you provision a cloud computing environment, optimizing for cost is a fundamental design tenant for architects. When selecting a solution, you should not only focus on the functional architecture and feature set but on the cost profile of the solutions you select.

AWS provides fine-grained billing, which enables you to track the costs associated with all aspects of your solutions. There are a range of services to help you manage budgets, alert you to costs incurred, and to help you optimize resource usage and costs.

Operations on AWS

When operating services on AWS, there are several common categories of operating models:

- Applications that are migrated, maintain existing traditional operating models, leverage the ability to manage Infrastructure as Code through APIs enabling robust and repeatable build processes, improving reliability.
- Solutions that are refactored leverage higher levels of automation of the operational processes as the supporting services, e.g. AWS Auto Scaling and self-healing architectures.
- Solutions that are rearchitected and designed for cloud operations are typically fully automated through DevOps processes for delivery pipeline and management.

Supporting these transitions does not just change the technologies used, but also cultural changes in the way that development and operational teams are managed.

AWS provides tooling, processes, and best practices to support the transition of operational practices to maximize the benefits that can be leveraged from cloud computing.

Design Principles

The AWS Cloud includes many design patterns and architectural options that you can apply to a wide variety of use cases. Some key design principles of the AWS Cloud include scalability, disposable resources, automation, loose coupling managed services instead of servers, and flexible data storage options.

Scalability

Systems that are expected to grow over time need to be built on top of a scalable architecture. Such an architecture can support growth in users, traffic, or data size with no drop-in performance. It should provide that scale in a linear manner where adding extra resources results in at least a proportional increase in ability to serve additional load. Growth should introduce economies of scale, and cost should follow the same dimension that generates business value out of that system. While cloud computing provides virtually unlimited on-demand capacity, your design needs to be able to take advantage of those resources seamlessly.

There are generally two ways to scale an IT architecture: vertically and horizontally.

Scaling Vertically

Scaling vertically takes place through an increase in the specifications of an individual resource, such as upgrading a server with a larger hard drive or a faster CPU. With Amazon EC2, you can stop an instance and resize it to an instance type that has more RAM, CPU, I/O, or networking capabilities. This way of scaling can eventually reach a limit, and it is not always a cost-efficient or highly available approach. However, it is very easy to implement and can be sufficient for many use cases especially in the short term.

Scaling Horizontally

Scaling horizontally takes place through an increase in the number of resources, such as adding more hard drives to a storage array or adding more servers to support an application. This is a great way to build internet-scale applications that leverage the elasticity of cloud computing. Not all architectures are designed to distribute their workload to multiple resources, so let's examine some possible scenarios.

Stateless Applications

When users or services interact with an application they will 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. For example, an application that, given the same input, provides the same response to any end user, is a *stateless application*. Stateless applications can scale horizontally because any of the available compute resources (such as EC2 instances and AWS Lambda functions) can service any request. Without stored session data, you can simply add more compute resources as needed. When that capacity is no longer required, you can safely terminate those individual resources, after running tasks have been drained. Those resources do not need to be aware of the presence of their peers—all that is required is a way to distribute the workload to them.

Distribute Load to Multiple Nodes

To distribute the workload to multiple nodes in your environment, you can choose either a *push* or a *pull* model.

With a *push* model, you can use Elastic Load Balancing (ELB) to distribute a workload. ELB routes incoming application requests across multiple EC2 instances. When routing traffic, a Network Load Balancer operates at layer 4 of the Open Systems Interconnection (OSI) model to handle millions of requests per second. With the adoption of container-based services, you can also use an Application Load Balancer. An Application Load Balancer provides Layer 7 of the OSI model and supports content-based routing of requests based on application traffic. Alternatively, you can use Amazon Route 53 to implement a DNS round robin. In this case, DNS responses return an IP address from a list of valid hosts in a round-robin fashion. While easy to implement, this approach does not always work well with the elasticity of cloud computing. This is because even if you can set low time to live (TTL) values for your DNS records, caching DNS resolvers are outside the control of Amazon Route 53 and might not always respect your settings.

Instead of a load balancing solution, you can implement a *pull* model for asynchronous, event-driven workloads. In a pull model, tasks that need to be performed or data that needs to be processed can be stored as messages in a queue using Amazon Simple Queue Service (Amazon SQS) or as a streaming

data solution such as Amazon Kinesis. Multiple compute resources can then pull and consume those messages, processing them in a distributed fashion.

Stateless Components

In practice, most applications maintain some kind of state information. For example, web applications need to track whether a user is signed in so that personalized content can be presented based on previous actions. An automated multi-step process also needs to track previous activity to decide what its next action should be. You can still make a portion of these architectures stateless by not storing anything that needs to persist for more than a single request in the local file system.

For example, web applications can use HTTP cookies to store session information (such as shopping cart items) in the web client cache. The browser passes that information back to the server at each subsequent request so that the application does not need to store it. However, this approach has two drawbacks. First, the content of the HTTP cookies can be tampered with on the client side, so you should always treat it as untrusted data that must be validated. Second, HTTP cookies are transmitted with every request, which means that you should keep their size to a minimum to avoid unnecessary latency.

Consider only storing a unique session identifier in an HTTP cookie and storing more detailed user session information on the server side. Most programming platforms provide a native session management mechanism that works this way. However, user session information is often stored on the local file system by default and results in a stateful architecture. A common solution to this problem is to store this information in a database. Amazon DynamoDB is a great choice because of its scalability, high availability, and durability characteristics. For many platforms, there are open source drop-in replacement libraries that allow you to store native sessions in Amazon DynamoDB.[4]

Other scenarios require storage of larger files (such as user uploads and interim results of batch processes). By placing those files in a shared storage layer such as Amazon Simple Storage Service (Amazon S3) or Amazon Elastic File System (Amazon EFS), you can avoid the introduction of stateful components.

Finally, a complex multi-step workflow is another example where you must track the current state of each execution. You can use AWS Step Functions to centrally store execution history and make these workloads stateless.

Stateful Components

Inevitably, there will be layers of your architecture that you won't turn into stateless components. By definition, databases are stateful. For more information, see the Databases section. In addition, many legacy applications were designed to run on a single server by relying on local compute resources. Other use cases might require client devices to maintain a connection to a specific server for prolonged periods. For example, real-time multiplayer gaming must offer multiple players a consistent view of the game world with very low latency. This is much simpler to achieve in a non-distributed implementation where participants are connected to the same server.

You might still be able to scale those components horizontally by distributing the load to multiple nodes with session affinity. In this model, you bind all the transactions of a session to a specific compute resource. But, this model does have some limitations. Existing sessions do not directly benefit from the introduction of newly launched compute nodes. More importantly, session affinity cannot be guaranteed. For example, when a node is terminated or becomes unavailable, users bound to it will be disconnected and experience a loss of session-specific data, which is anything that is not stored in a shared resource such as Amazon S3, Amazon EFS, or a database.

Implement Session Affinity

For HTTP and HTTPS traffic, you can use the *sticky sessions* feature of an Application Load Balancer to bind a user's session to a specific instance.[5] With this feature, an Application Load Balancer will try to use the same server for that user for the duration of the session.

Another option—if you control the code that runs on the client—is to use client-side load balancing. This adds extra complexity, but can be useful in scenarios where a load balancer does not meet your requirements. For example, you might be using a protocol that's not supported by ELB, or you might need full control over how users are assigned to servers (e.g., in a gaming scenario, you might need to make sure that game participants are matched and connect to the same server). In this model, the clients need a way of discovering valid server endpoints to directly connect to. You can use DNS for that, or you can build a

simple discovery API to provide that information to the software running on the client. In the absence of a load balancer, the health-checking mechanism also needs to be implemented on the client side. You should design your client logic so that when server unavailability is detected, devices reconnect to another server with little disruption for the application.

Distributed Processing

Use cases that involve the processing of very large amounts of data—anything that can't be handled by a single compute resource in a timely manner—require a distributed processing approach. By dividing a task and its data into many small fragments of work, you can execute them in parallel across a set of compute resources.

Implement Distributed Processing

Offline batch jobs can be horizontally scaled by using distributed data processing engines such as AWS Batch, AWS Glue, and Apache Hadoop. On AWS, you can use Amazon EMR to run Hadoop workloads on top of a fleet of EC2 instances without the operational complexity. For real-time processing of streaming data, Amazon Kinesis partitions data in multiple shards that can then be consumed by multiple Amazon EC2 or AWS Lambda resources to achieve scalability.

For more information on these types of workloads, see the Big Data Analytics Options on AWS[6] and Core Tenets of IoT whitepapers.[7]

Disposable Resources Instead of Fixed Servers

In a traditional infrastructure environment, you have to work with fixed resources because of the upfront cost and lead time of introducing new hardware. This drives practices such as manually logging in to servers to configure software or fix issues, hardcoding IP addresses, and running tests or processing jobs sequentially.

When designing for AWS, you can take advantage of the dynamically provisioned nature of cloud computing. You can think of servers and other components as temporary resources. You can launch as many as you need, and use them only for as long as you need them.

Another issue with fixed, long-running servers is *configuration drift*. Changes and software patches applied through time can result in untested and heterogeneous configurations across different environments. You can solve this problem with an immutable infrastructure pattern. With this approach, a server—once launched—is never updated. Instead, when there is a problem or need for an update, the problem server is replaced with a new server that has the latest configuration. This enables resources to always be in a consistent (and tested) state, and makes rollbacks easier to perform. This is more easily supported with stateless architectures.

Instantiating Compute Resources

Whether you are deploying a new environment for testing, or increasing capacity of an existing system to cope with extra load, you do not want to manually set up new resources with their configuration and code. It is important that you make this an automated and repeatable process that avoids long lead times and is not prone to human error. There are a few methods to achieve this.

Bootstrapping

When you launch an AWS resource such as an EC2 instance or Amazon Relational Database Service (Amazon RDS) DB instance, you start with a default configuration. You can then execute automated bootstrapping actions, which are scripts that install software or copy data to bring that resource to a particular state. You can parameterize configuration details that vary between different environments (such as production or test) so that you can reuse the same scripts without modifications.

You can set up new EC2 instances with user data scripts and cloud-init directives.[8] You can use simple scripts and configuration management tools such as Chef or Puppet. In addition, with custom scripts and the AWS APIs, or with AWS CloudFormation support for AWS Lambda-backed custom resources, you can write provisioning logic that acts on almost any AWS resource.[9]

Golden Images

Certain AWS resource types, such as EC2 instances, Amazon RDS DB instances, and Amazon Elastic Block Store (Amazon EBS) volumes, can be launched from a *golden image*, which is a snapshot of a particular state of that resource. When compared to the bootstrapping approach, a golden image results in faster start times and removes dependencies to configuration services or third-party

repositories. This is important in auto-scaled environments where you want to be able to quickly and reliably launch additional resources as a response to demand changes.

You can customize an EC2 instance and then save its configuration by creating an Amazon Machine Image (AMI).[10] You can launch as many instances from the AMI as you need, and they will all include those customizations. Each time you want to change your configuration you must create a new golden image, so you must have a versioning convention to manage your golden images over time. We recommend that you use a script to create the bootstrap for the EC2 instances that you use to create your AMIs. This gives you a flexible method to test and modify those images through time.

Alternatively, if you have an existing on-premises virtualized environment, you can use VM Import/Export from AWS to convert a variety of virtualization formats to an AMI. You can also find and use prebaked, shared AMIs provided either by AWS or third parties in AWS Marketplace.

While golden images are most commonly used when you launch new EC2 instances, they can also be applied to resources such as Amazon RDS DB instances or Amazon EBS volumes. For example, when you launch a new test environment, you might want to prepopulate its database by instantiating it from a specific Amazon RDS snapshot, instead of importing the data from a lengthy SQL script.

Containers

Another option popular with developers is Docker—an open-source technology that allows you to build and deploy distributed applications inside software containers. Docker allows you to package a piece of software in a Docker image, which is a standardized unit for software development, containing everything the software needs to run: code, runtime, system tools, system libraries, etc. AWS Elastic Beanstalk, Amazon Elastic Container Service (Amazon ECS) and AWS Fargate let you deploy and manage multiple containers across a cluster of EC2 instances. You can build golden Docker images and use the ECS Container Registry to manage them.

An alternative container environment is Kubernetes and Amazon Elastic Container Service for Kubernetes (Amazon EKS). With Kubernetes and Amazon EKS, you can easily deploy, manage, and scale containerized applications.

Hybrid

You can also use a combination of the two approaches: some parts of the configuration are captured in a golden image, while others are configured dynamically through a bootstrapping action.

Items that do not change often or that introduce external dependencies will typically be part of your golden image. An example of a good candidate is your web server software that would otherwise have to be downloaded by a third-party repository each time you launch an instance.

Items that change often or differ between your various environments can be set up dynamically through bootstrapping actions. For example, if you are deploying new versions of your application frequently, creating a new AMI for each application version might be impractical. You also do not want to hard code the database hostname configuration to your AMI because that would be different between the test and production environments. User data or tags allow you to use more generic AMIs that can be modified at launch time. For example, if you run web servers for various small businesses, they can all use the same AMI and retrieve their content from an S3 bucket location that you specify in the user data at launch.

AWS Elastic Beanstalk follows the hybrid model. It provides preconfigured run time environments—each initiated from its own AMI[11]—but allows you to run bootstrap actions through .ebextensions configuration files[12], and configure environmental variables to parameterize the environment differences.

For a more detailed discussion of the different ways you can manage deployments of new resources, see the Overview of Deployment Options on AWS[13] and Managing Your AWS Infrastructure at Scale whitepapers.[14]

Infrastructure as Code

Application of the principles we have discussed does not have to be limited to the individual resource level. Because AWS assets are programmable, you can apply techniques, practices, and tools from software development to make your whole infrastructure reusable, maintainable, extensible, and testable. For more information, see the Infrastructure as Code whitepaper.[15]

AWS CloudFormation templates give you an easy way to create and manage a collection of related AWS resources, and provision and update them

in an orderly and predictable fashion. You can describe the AWS resources and any associated dependencies or runtime parameters required to run your application. Your CloudFormation templates can live with your application in your version control repository, which allows you to reuse architectures and reliably clone production environments for testing.

Automation

In a traditional IT infrastructure, you often have to manually react to a variety of events. When deploying on AWS, there is an opportunity for automation, so that you improve both your system's stability and the efficiency of your organization. Consider introducing one or more of these types of automation into your application architecture to ensure more resiliency, scalability, and performance.

Serverless Management and Deployment

When you adopt serverless patterns, the operational focus is on the automation of the deployment pipeline. AWS manages the underlying services, scale, and availability. AWS CodePipeline, AWS CodeBuild, and AWS CodeDeploy support the automation of the deployment of these processes.[16]

Infrastructure Management and Deployment

AWS Elastic Beanstalk: You can use this service to deploy and scale web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS.[17] Developers can simply upload their application code, and the service automatically handles all the details, such as resource provisioning, load balancing, auto scaling, and monitoring.

Amazon EC2 auto recovery: You can create an Amazon CloudWatch alarm that monitors an EC2 instance and automatically recovers it if it becomes impaired.[18] A recovered instance is identical to the original instance, including the instance ID, private IP addresses, Elastic IP addresses, and all instance metadata. However, this feature is only available for applicable instance configurations. Refer to the Amazon EC2 documentation for an up-to-date description of those preconditions. In addition, during instance recovery, the instance is migrated through an instance reboot, and any data that is in-memory is lost.

AWS Systems Manager: You can automatically collect software inventory, apply OS patches, create a system image to configure Windows and Linux operating systems, and execute arbitrary commands. Provisioning these services simplifies the operating model and ensures the optimum environment configuration.[19]

Auto Scaling: You can maintain application availability and scale your Amazon EC2, Amazon DynamoDB, Amazon ECS, Amazon Elastic Container Service for Kubernetes (Amazon EKS) capacity up or down automatically according to the conditions you define.[20] You can use Auto Scaling to help make sure that you are running the desired number of healthy EC2 instances across multiple Availability Zones. Auto Scaling can also automatically increase the number of EC2 instances during demand spikes to maintain performance and decrease capacity during less busy periods to optimize costs.

Alarms and Events

Amazon CloudWatch alarms: You can create a CloudWatch alarm that sends an Amazon Simple Notification Service (Amazon SNS) message when a particular metric goes beyond a specified threshold for a specified number of periods.[21] Those Amazon SNS messages can automatically kick off the execution of a subscribed Lambda function, enqueue a notification message to an Amazon SQS queue, or perform a POST request to an HTTP or HTTPS endpoint.

Amazon CloudWatch Events: Delivers a near real-time stream of system events that describe changes in AWS resources.[22] Using simple rules, you can route each type of event to one or more targets, such as Lambda functions, Kinesis streams, and SNS topics.

AWS Lambda scheduled events: You can create a Lambda function and configure AWS Lambda to execute it on a regular schedule.[23]

AWS WAF security automations: AWS WAF is a web application firewall that enables you to create custom, application-specific rules that block common attack patterns that can affect application availability, compromise security, or consume excessive resources. You can administer AWS WAF completely through APIs, which makes security automation easy, enabling rapid rule propagation and fast incident response.[24]

Loose Coupling

As application complexity increases, a desirable attribute of an IT system is that it can be broken into smaller, loosely coupled components. This means that IT systems should be designed in a way that reduces interdependencies—a change or a failure in one component should not cascade to other components.

Well-Defined Interfaces

A way to reduce interdependencies in a system is to allow the various components to interact with each other only through specific, technology-agnostic interfaces, such as RESTful APIs. In that way, technical implementation detail is hidden so that teams can modify the underlying implementation without affecting other components. As long as those interfaces maintain backwards compatibility, deployments of difference components are decoupled. This granular design pattern is commonly referred to as a microservices architecture.

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

Service Discovery

Applications that are deployed as a set of smaller services depend on the ability of those services to interact with each other. Because each of those services can be running across multiple compute resources, there needs to be a way for each service to be addressed. For example, in a traditional infrastructure, if your front-end web service needs to connect with your back-end web service, you could hardcode the IP address of the compute resource where this service was running. Although this approach can still work in cloud computing, if those services are meant to be loosely coupled, they should be able to be consumed without prior knowledge of their network topology details. Apart from hiding complexity, this also allows infrastructure details to change at any time. Loose coupling is a crucial element if you want to take advantage of the elasticity of cloud computing where new resources can be launched or terminated at any point in time. In order to achieve that you will need some way of implementing service discovery.

Implement Service Discovery

For an Amazon EC2-hosted service, a simple way to achieve service discovery is through Elastic Load Balancing (ELB). Because each load balancer gets its own hostname, you can consume a service through a stable endpoint. This can be combined with DNS and private Amazon Route 53 zones, so that the particular load balancer's endpoint can be abstracted and modified at any time.

Another option is to use a service registration and discovery method to allow retrieval of the endpoint IP addresses and port number of any service. Because service discovery becomes the glue between the components, it is important that it is highly available and reliable. If load balancers are not used, service discovery should also allow options such as health checks. Amazon Route 53 supports auto naming to make it easier to provision instances for microservices. Auto naming lets you automatically create DNS records based on a configuration you define. Other example implementations include custom solutions using a combination of tags, a highly available database, custom scripts that call the AWS APIs, or open-source tools such as Netflix Eureka, Airbnb Synapse, or HashiCorp Consul.

Asynchronous Integration

Asynchronous integration is another form of loose coupling between services. This model is suitable for any interaction that does not need an immediate response and where an acknowledgement that a request has been registered will suffice. It involves one component that generates events and another that consumes them. The two components do not integrate through direct point-to-point interaction but usually through an intermediate durable storage layer, such as an SQS queue or a streaming data platform such as Amazon Kinesis, cascading Lambda events, AWS Step Functions, or Amazon Simple Workflow Service.

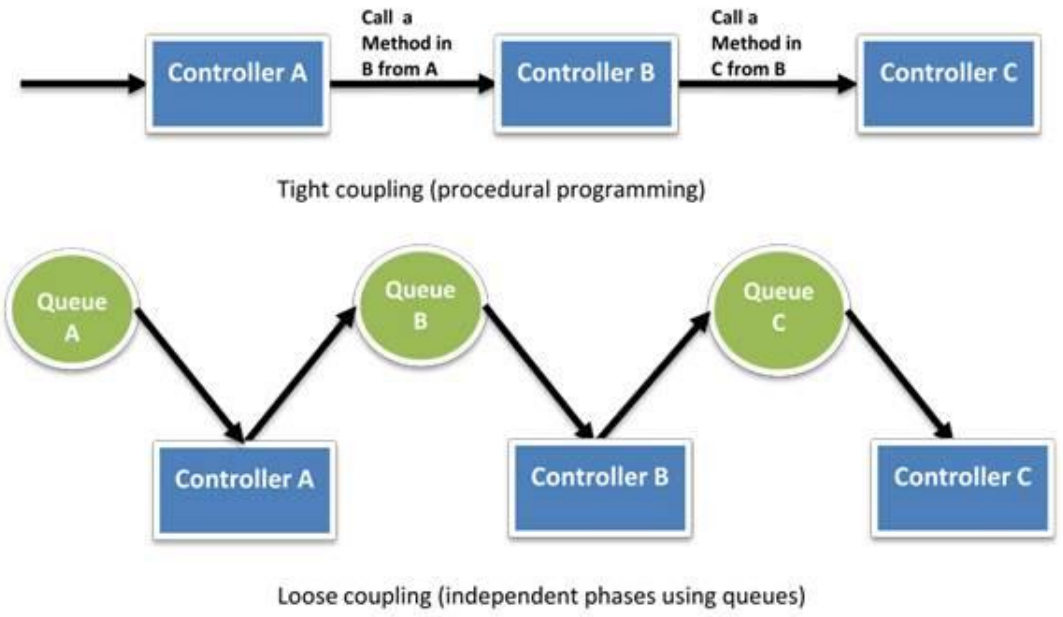

Figure 1: Tight and loose coupling

This approach decouples the two components and introduces additional resiliency. So, for example, if a process that is reading messages from the queue fails, messages can still be added to the queue and processed when the system recovers. It also allows you to protect a less scalable back-end service from front-end spikes and find the right tradeoff between cost and processing lag. For example, you can decide that you don't need to scale your database to accommodate an occasional peak of write queries, as long as you eventually process those queries asynchronously with some delay. Finally, by removing slow operations from interactive request paths you can also improve the end-user experience.

Examples of asynchronous integration include:

- A front-end application inserts jobs in a queue system such as Amazon SQS. A back-end system retrieves those jobs and processes them at its own pace.
- An API generates events and pushes them into Kinesis streams. A back-end application processes these events in batches to create aggregated time-series data stored in a database.

- Multiple heterogeneous systems use AWS Step Functions to communicate the flow of work between them without directly interacting with each other.
- Lambda functions can consume events from a variety of AWS sources, such as Amazon DynamoDB update streams and Amazon S3 event notifications. You don't have to worry about implementing a queuing or other asynchronous integration method because Lambda handles this for you.

Distributed Systems Best Practices

Another way to increase loose coupling is to build applications that handle component failure in a graceful manner. You can identify ways to reduce the impact to your end users and increase your ability to make progress on your offline procedures, even in the event of some component failure.

Graceful Failure in Practice

A request that fails can be retried with an exponential backoff and Jitter strategy, or it can be stored in a queue for later processing.[25] For front-end interfaces, it might be possible to provide alternative or cached content instead of failing completely when, for example, your database server becomes unavailable. The Amazon Route 53 DNS failover feature also gives you the ability to monitor your website and automatically route your visitors to a backup site if your primary site becomes unavailable. You can host your backup site as a static website on Amazon S3 or as a separate dynamic environment.

Services, Not Servers

Developing, managing, and operating applications, especially at scale, requires a wide variety of underlying technology components. With traditional IT infrastructure, companies would have to build and operate all those components.

AWS offers a broad set of compute, storage, database, analytics, application, and deployment services that help organizations move faster and lower IT costs.

Architectures that do not leverage that breadth (e.g., if they use only Amazon EC2) might not be making the most of cloud computing and might be missing an opportunity to increase developer productivity and operational efficiency.

Managed Services

AWS managed services provide building blocks that developers can consume to power their applications. These managed services include databases, machine learning, analytics, queuing, search, email, notifications, and more. For example, with Amazon SQS you can offload the administrative burden of operating and scaling a highly available messaging cluster, while paying a low price for only what you use. Amazon SQS is also inherently scalable and reliable. The same applies to Amazon S3, which enables you to store as much data as you want and access it when you need it, without having to think about capacity, hard disk configurations, replication, and other related issues.

Other examples of managed services that power your applications include:[26]

- Amazon CloudFront for content delivery
- ELB for load balancing
- Amazon DynamoDB for NoSQL databases
- Amazon CloudSearch for search workloads
- Amazon Elastic Transcoder for video encoding
- Amazon Simple Email Service (Amazon SES) for sending and receiving emails[27]

Serverless Architectures

Serverless architectures can reduce the operational complexity of running applications. It is possible to build both event-driven and synchronous services for mobile, web, analytics, CDN business logic, and IoT without managing any server infrastructure. These architectures can reduce costs because you don't have to manage or pay for underutilized servers, or provision redundant infrastructure to implement high availability.

For example, you can upload your code to the AWS Lambda compute service, and the service can run the code on your behalf using AWS infrastructure. With AWS Lambda, you are charged for every 100ms your code executes and the number of times your code is triggered. By using Amazon API Gateway, you can develop virtually infinitely scalable synchronous APIs powered by AWS Lambda. When combined with Amazon S3 for serving static content assets, this pattern can deliver a complete web application. For more details on this type of architecture, see the AWS Serverless Multi-Tier Architectures whitepaper.[28]

When it comes to mobile and web apps, you can use Amazon Cognito so that you don't have to manage a back-end solution to handle user authentication, network state, storage, and sync. Amazon Cognito generates unique identifiers for your users.

Those identifiers can be referenced in your access policies to enable or restrict access to other AWS resources on a per-user basis. Amazon Cognito provides temporary AWS credentials to your users, allowing the mobile application running on the device to interact directly with AWS Identity and Access Management (IAM)-protected AWS services. For example, using IAM you can restrict access to a folder in an S3 bucket to a particular end user.

For IoT applications, organizations have traditionally had to provision, operate, scale, and maintain their own servers as device gateways to handle the communication between connected devices and their services. AWS IoT provides a fully managed device gateway that scales automatically with your usage without any operational overhead.

Serverless architectures have also made it possible to run responsive services at edge locations. AWS Lambda@Edge lets you run Lambda functions at Amazon CloudFront edge locations in response to CloudFront events. This enables patterns for low-latency solutions and the introduction of functionality without changing underlying applications.[29]

For data analytics, managing queries on large volumes of data typically requires you to manage a complex infrastructure. Amazon Athena is an interactive query service that makes it easy for you to analyze data in Amazon S3 using standard SQL. Athena is serverless, so there is no infrastructure to manage, and you pay only for the queries that you run.

Databases

With traditional IT infrastructure, organizations are often limited to the database and storage technologies they can use. There can be constraints based on licensing costs and the ability to support diverse database engines. On AWS, these constraints are removed by managed database services that offer enterprise performance at open-source cost. As a result, it is not uncommon for applications to run on top of a polyglot data layer choosing the right technology for each workload.

Choose the Right Database Technology for Each Workload

These questions can help you make decisions about which solutions to include in your architecture:

- Is this a read-heavy, write-heavy, or balanced workload? How many reads and writes per second are you going to need? How will those values change if the number of users increases?
- How much data will you need to store and for how long? How quickly will this grow? Is there an upper limit in the near future? What is the size of each object (average, min, max)? How will these objects be accessed?
- What are the requirements in terms of durability of data? Is this data store going to be your "source of truth?"
- What are your latency requirements? How many concurrent users do you need to support?
- What is your data model and how are you going to query the data? Are your queries relational in nature (e.g., JOINs between multiple tables)? Could you denormalize your schema to create flatter data structures that are easier to scale?
- What kind of functionality do you require? Do you need strong integrity controls, or are you looking for more flexibility (e.g., schema-less data stores)? Do you require sophisticated reporting or search capabilities? Are your developers more familiar with relational databases than NoSQL?
- What are the associated database technology license costs? Do these costs consider application development investment, storage, and usage costs over time? Does the licensing model support projected growth? Could you use cloud-native database engines such as Amazon Aurora to get the simplicity and cost-effectiveness of open-source databases?

Relational Databases

Relational databases (also known as RDBS or SQL databases) normalize data into well-defined tabular structures known as tables, which consist of rows and columns. They provide a powerful query language, flexible indexing capabilities, strong integrity controls, and the ability to combine data from multiple tables in a fast and efficient manner. Amazon RDS makes it easy to set up, operate, and scale a relational database in the cloud with support for many familiar database engines.

Scalability

Relational databases can scale vertically by upgrading to a larger Amazon RDS DB instance or adding more and faster storage. In addition, consider using Amazon Aurora, which is a database engine designed to deliver much higher throughput compared to standard MySQL running on the same hardware. For read-heavy applications, you can also horizontally scale beyond the capacity constraints of a single DB instance by creating one or more read replicas.

Read replicas are separate database instances that are replicated asynchronously. As a result, they are subject to replication lag and might be missing some of the latest transactions. Application designers need to consider which queries have tolerance to slightly stale data. Those queries can be executed on a read replica, while the remainder should run on the primary node. Read replicas can also not accept any write queries.

Relational database workloads that need to scale their write capacity beyond the constraints of a single DB instance require a different approach called *data partitioning* or *sharding*. With this model, data is split across multiple database schemas each running in its own autonomous primary DB instance. Although Amazon RDS removes the operational overhead of running those instances, sharding introduces some complexity to the application. The application's data access layer needs to be modified to have awareness of how data is split so that it can direct queries to the right instance. In addition, schema changes must be performed across multiple database schemas, so it is worth investing some effort to automate this process.

High Availability

For any production relational database, we recommend using the Amazon RDS Multi-AZ deployment feature, which creates a synchronously replicated standby instance in a different Availability Zone. In case of failure of the primary node, Amazon RDS performs an automatic failover to the standby without the need for manual administrative intervention. When a failover is performed, there is a short period during which the primary node is not accessible. Resilient applications can be designed for Graceful Failure by offering reduced functionality, such as read-only mode by using read replicas. Amazon Aurora provides multi-master capability to enable reads and writes to be scaled across Availability Zones and also supports cross-Region replication.

Anti-Patterns

If your application primarily indexes and queries data with no need for joins or complex transactions—especially if you expect a write throughput beyond the constraints of a single instance—consider a NoSQL database instead. If you have large binary files (audio, video, and image), it will be more efficient to store the actual files in Amazon S3 and only hold the metadata for the files in your database.

For more detailed relational database best practices, see the Amazon RDS documentation.[30]

NoSQL Databases

NoSQL databases trade some of the query and transaction capabilities of relational databases for a more flexible data model that seamlessly scales horizontally. NoSQL databases use a variety of data models, including graphs, key-value pairs, and JSON documents, and are widely recognized for ease of development, scalable performance, high availability, and resilience. Amazon DynamoDB is a fast and flexible NoSQL database service for applications that need consistent, single-digit, millisecond latency at any scale.[31] It is a fully managed cloud database and supports both document and key-value store models.

Scalability

NoSQL database engines will typically perform data partitioning and replication to scale both the reads and the writes in a horizontal fashion. They do this transparently, and don't need the data partitioning logic implemented in the data access layer of your application. Amazon DynamoDB in particular manages table partitioning automatically, adding new partitions as your table grows in size or as read-provisioned and write-provisioned capacity changes. Amazon DynamoDB Accelerator (DAX) is a managed, highly available, in-memory cache for DynamoDB to leverage significant performance improvements.[32]

To learn about how you can make the most of Amazon DynamoDB scalability when you design your application, see Best Practices for DynamoDB.[33]

High Availability

Amazon DynamoDB synchronously replicates data across three facilities in an AWS Region, which provides fault tolerance in the event of a server failure or Availability Zone disruption. Amazon DynamoDB also supports global tables to provide a fully managed, multi-Region, multi-master database that provides fast, local, read-and-write performance for massively scaled global applications. Global Tables are replicated across your selected AWS Regions.

Anti-Patterns

If your schema cannot be denormalized, and your application requires joins or complex transactions, consider a relational database instead. If you have large binary files (audio, video, and image), consider storing the files in Amazon S3 and storing the metadata for the files in your database.

For guidance on migrating from a relational database to DynamoDB, or on evaluating which workloads to migrate, see the Best Practices for Migrating from RDBMS to DynamoDB whitepaper.[34]

Data Warehouse

A data warehouse is a specialized type of relational database, which is optimized for analysis and reporting of large amounts of data. It can be used to combine transactional data from disparate sources (such as user behavior in a web application, data from your finance and billing system, or customer relationship management or CRM) to make them available for analysis and decision-making.

Traditionally, setting up, running, and scaling a data warehouse has been complicated and expensive. On AWS, you can leverage Amazon Redshift, a managed data warehouse service that is designed to operate at less than a tenth the cost of traditional solutions.

Scalability

Amazon Redshift achieves efficient storage and optimum query performance through a combination of massively parallel processing (MPP), columnar data storage, and targeted data compression encoding schemes. It is particularly suited to analytic and reporting workloads against very large data sets. The Amazon Redshift MPP architecture enables you to increase performance by increasing the number of nodes in your data warehouse cluster. Amazon Redshift Spectrum enables Amazon Redshift SQL queries against exabytes of data in Amazon S3, which extends the analytic capabilities of Amazon Redshift beyond data stored on local disks in the data warehouse to unstructured data, without the need to load or transform data.

High Availability

Amazon Redshift has multiple features that enhance the reliability of your data warehouse cluster. We recommend that you deploy production workloads in multi-node clusters, so that data that is written to a node is automatically replicated to other nodes within the cluster. Data is also continuously backed up to Amazon S3. Amazon Redshift continuously monitors the health of the cluster and automatically re-replicates data from failed drives and replaces nodes as necessary. For more information, see the Amazon Redshift FAQ.[35]

Anti-Patterns

Because Amazon Redshift is an SQL-based relational database management system (RDBMS), it is compatible with other RDBMS applications and business intelligence tools. Although Amazon Redshift provides the functionality of a typical RDBMS, including online transaction processing (OLTP) functions, it is not designed for these workloads. If you expect a high concurrency workload

that generally involves reading and writing all of the columns for a small number of records at a time, you should instead consider using Amazon RDS or Amazon DynamoDB.

Search

Search is often confused with query. A query is a formal database query, which is addressed in formal terms to a specific data set. Search enables datasets to be queried that are not precisely structured. For this reason, applications that require sophisticated search functionality will typically outgrow the capabilities of relational or NoSQL databases. A search service can be used to index and search both structured and free text format and can support functionality that is not available in other databases, such as customizable result ranking, faceting for filtering, synonyms, and stemming.

On AWS, you can choose between Amazon CloudSearch and Amazon Elasticsearch Service (Amazon ES). Amazon CloudSearch is a managed service that requires little configuration and will scale automatically. Amazon ES offers an open-source API and gives you more control over the configuration details. Amazon ES has also evolved to become more than just a search solution. It is often used as an analytics engine for use cases such as log analytics, real-time application monitoring, and click stream analytics.

Scalability

Both Amazon CloudSearch and Amazon ES use data partitioning and replication to scale horizontally. The difference is that with Amazon CloudSearch, you don't need to worry about how many partitions and replicas you need because the service automatically handles that.

High Availability

Both Amazon CloudSearch and Amazon ES include features that store data redundantly across Availability Zones. For details, see the Amazon CloudSearch[36] and Amazon ES documentation.[37]

Graph Databases

A graph database uses graph structures for queries. A graph is defined as consisting of edges (relationships), which directly relate to nodes (data entities) in the store. The relationships enable data in the store to be linked together directly, which allows for the fast retrieval of complex hierarchical structures in relational systems. For this reason, graph databases are purposely built to store

and navigate relationships and are typically used in use cases like social networking, recommendation engines, and fraud detection, in which you need to be able to create relationships between data and quickly query these relationships.

Amazon Neptune is a fully-managed graph database service. For more information, see the Amazon Neptune FAQ.

Scalability

Amazon Neptune is a purpose-built, high-performance graph database optimized for processing graph queries.

High Availability

Amazon Neptune is highly available, with read replicas, point-in-time recovery, continuous backup to Amazon S3, and replication across Availability Zones. Neptune is secure, with support for encryption at rest and in transit.[38]

Managing Increasing Volumes of Data

Traditional data storage and analytics tools can no longer provide the agility and flexibility required to deliver relevant business insights. That's why many organizations are shifting to a data lake architecture. A data lake is an architectural approach that allows you to store massive amounts of data in a central location so that it's readily available to be categorized, processed, analyzed, and consumed by diverse groups within your organization. Since data can be stored as-is, you do not have to convert it to a predefined schema, and you no longer need to know what questions to ask about your data beforehand. This enables you to select the correct technology to meet your specific analytical requirements. For more information, see the Building a Data Lake with Amazon Web Services whitepaper.[39]

Removing Single Points of Failure

Production systems typically come with defined or implicit objectives for uptime. A system is highly available when it can withstand the failure of an individual component or multiple components, such as hard disks, servers, and network links. To help you create a system with high availability, you can think about ways to automate recovery and reduce disruption at every layer of your architecture. For more information about high availability design patterns, refer to the Building Fault Tolerant Applications whitepaper.[40]

Introducing Redundancy

Single points of failure can be removed by introducing redundancy, which means you have multiple resources for the same task. Redundancy can be implemented in either standby or active mode.

In *standby redundancy*, when a resource fails, functionality is recovered on a secondary resource with the failover process. The failover typically requires some time before it completes, and during this period the resource remains unavailable. The secondary resource can either be launched automatically only when needed (to reduce cost), or it can already be running idle (to accelerate failover and minimize disruption). Standby redundancy is often used for stateful components such as relational databases.

In *active redundancy*, requests are distributed to multiple redundant compute resources. When one of them fails, the rest can simply absorb a larger share of the workload. Compared to standby redundancy, active redundancy can achieve better usage and affect a smaller population when there is a failure.

Detect Failure

You should aim to build as much automation as possible in both detecting and reacting to failure. You can use services such as ELB and Amazon Route 53 to configure health checks and mask failure by routing traffic to healthy endpoints. In addition, you can replace unhealthy nodes automatically using Auto Scaling or by using the Amazon EC2 auto-recovery feature or services such as AWS Elastic Beanstalk.[41] It won't be possible to predict every possible failure scenario on day one. Make sure you collect enough logs and metrics to understand normal system behavior. After you understand that, you will be able to set up alarms for manual intervention or automated response.

Design Good Health Checks

Configuring the right health checks for your application helps determine your ability to respond correctly and promptly to a variety of failure scenarios. Specifying the wrong health check can actually reduce your application's availability.

In a typical three-tier application, you configure health checks on ELB. Design your health checks with the objective of reliably assessing the health of the back-end nodes. A simple TCP health check won't detect if the instance itself is healthy but the web server process has crashed. Instead, you should assess whether the web server can return an HTTP 200 response for some simple request.

At this layer, it might not be a good idea to configure a *deep health check*, which is a test that depends on other layers of your application to be successful, because false positives can result. For example, if your health check also assesses whether the instance can connect to a back-end database, you risk marking all of your web servers as unhealthy when that database node becomes shortly unavailable. A layered approach is often the best. A deep health check might be appropriate to implement at the Amazon Route 53 level. By running a more holistic check that determines if that environment is able to actually provide the required functionality, you can configure Amazon Route 53 to failover to a static version of your website until your database is up and running again.

Durable Data Storage

Your application and your users will create and maintain a variety of data. It is crucial that your architecture protects both data availability and integrity. Data replication is the technique that introduces redundant copies of data. It can help horizontally scale read capacity, but it also increases data durability and availability. Replication can occur in a few different modes.

Synchronous replication only acknowledges a transaction after it has been durably stored in both the primary location and its replicas. It is ideal for protecting the integrity of data from the event of a failure of the primary node. Synchronous replication can also scale read capacity for queries that require the most up-to-date data (strong consistency). The drawback of synchronous replication is that the primary node is coupled to the replicas. A transaction can't be acknowledged before all replicas have performed the write. This can

compromise performance and availability, especially in topologies that run across unreliable or high-latency network connections. For the same reason, it is not recommended to maintain many synchronous replicas.

Regardless of the durability of your solution, this is no replacement for backups. Synchronous replication redundantly stores all updates to your data—even those that are results of software bugs or human error. However, particularly for objects stored on Amazon S3, you can use versioning to preserve, retrieve, and restore any of their versions.[42] With versioning, you can recover from both unintended user actions and application failures.

Asynchronous replication decouples the primary node from its replicas at the expense of introducing replication lag. This means that changes on the primary node are not immediately reflected on its replicas. Asynchronous replicas are used to horizontally scale the system's read capacity for queries that can tolerate that replication lag. It can also be used to increase data durability when some loss of recent transactions can be tolerated during a failover. For example, you can maintain an asynchronous replica of a database in a separate AWS Region as a disaster recovery solution.

Quorum-based replication combines synchronous and asynchronous replication to overcome the challenges of large-scale distributed database systems.

Replication to multiple nodes can be managed by defining a minimum number of nodes that must participate in a successful write operation. A detailed discussion of distributed data stores is beyond the scope of this document. You can refer to For more information about distributed data stores and the core set of principles for an ultra-scalable and highly reliable database system, see the Amazon Dynamo whitepaper.[43]

It is important to understand where each technology you are using fits in these data storage models. Their behavior during various failover or backup/restore scenarios should align to your recovery point objective (RPO) and your recovery time objective (RTO). You must ascertain how much data you expect to lose and how quickly you need to resume operations. For example, the Redis engine for Amazon ElastiCache supports replication with automatic failover, but the Redis engine's replication is asynchronous. During a failover, it is highly likely that some recent transactions will be lost. However, Amazon RDS, with its Multi-AZ

feature, is designed to provide synchronous replication to keep data on the standby node up-to-date with the primary.

Automated Multi-Data Center Resilience

Business-critical applications also need protection against disruption scenarios that affect more than just a single disk, server, or rack. In a traditional infrastructure, you typically have a disaster recovery plan to allow failover to a distant second data center in the event of a major disruption in the primary one. Because of the long distance between the two data centers, latency makes it impractical to maintain synchronous cross-data center copies of the data. As a result, a failover will most certainly lead to data loss or a very costly data recovery process. This makes failover a risky and not always sufficiently tested procedure. Nevertheless, this is a model that provides excellent protection against a low probability but huge impact risk, such as a natural catastrophe that brings down your whole infrastructure for a long time. For more guidance on how to implement this approach on AWS, see the AWS Disaster Recovery whitepaper.[44]

A shorter interruption in a data center is a more likely scenario. For short disruptions in which the duration of the failure isn't predicted to be long, the choice to perform a failover is a difficult one and is generally avoided. On AWS, it is possible to adopt a simpler, more efficient protection from this type of failure. Each AWS Region contains multiple distinct locations, or *Availability Zones*. Each Availability Zone is engineered to be independent from failures in other Availability Zones. An Availability Zone is a data center, and in some cases, an Availability Zone consists of multiple data centers. Availability Zones within a Region provide inexpensive, low-latency network connectivity to other zones in the same Region. This allows you to replicate your data across data centers in a synchronous manner so that failover can be automated and be transparent for your users.

It is also possible to implement active redundancy. For example, a fleet of application servers can be distributed across multiple Availability Zones and be attached to ELB. When the EC2 instances of a particular Availability Zone fail their health checks, ELB stops sending traffic to those nodes. In addition, AWS Auto Scaling ensures that the correct number of EC2 instances are available to run your application, launching and terminating instances based on demand and defined by your scaling policies. If your application requires no short-term performance degradation because of an Availability Zone failure, your

architecture should be statically stable, which means it does not require a change in the behavior of your workload to tolerate failures. In this scenario, your architecture should provision excess capacity to withstand the loss of one Availability Zone. [45]

Many of the higher-level services on AWS are inherently designed according to the multiple Availability Zone (Multi-AZ) principle. For example, Amazon RDS provides high availability and automatic failover support for DB instances using Multi-AZ deployments, while with Amazon S3 and Amazon DynamoDB your data is redundantly stored across multiple facilities.

Fault Isolation and Traditional Horizontal Scaling

Though the active redundancy pattern is great for balancing traffic and handling instance or Availability Zone disruptions, it is not sufficient if there is something harmful about the requests themselves. For example, there could be scenarios where every instance is affected. If a particular request happens to trigger a bug that causes the system to fail over, then the caller may trigger a cascading failure by repeatedly trying the same request against all instances.

Shuffle Sharding

One fault-isolating improvement you can make to traditional horizontal scaling is called *sharding*. Similar to the technique traditionally used with data storage systems, instead of spreading traffic from all customers across every node, you can group the instances into shards. For example, if you have eight instances for your service, you might create four shards of two instances each (two instances for some redundancy within each shard) and distribute each customer to a specific shard. In this way, you are able to reduce the impact on customers in direct proportion to the number of shards you have. However, some customers will still be affected, so the key is to make the client fault tolerant. If the client can try every endpoint in a set of sharded resources until one succeeds, you get a dramatic improvement. This technique is known as *shuffle sharding*. For more information about this technique see the Shuffle Sharding: Massive and Magical Fault Isolation blog post.[46]

Optimize for Cost

When you move your existing architectures into the cloud, you can reduce capital expenses and drive savings as a result of the AWS economies of scale. By iterating and using more AWS capabilities, you can realize further opportunity to create cost-optimized cloud architectures. For more information about how to optimize for cost with AWS cloud computing, see the Cost Optimization with AWS whitepaper.[47]

Right Sizing

AWS offers a broad range of resource types and configurations for many use cases. For example, services such as Amazon EC2, Amazon RDS, Amazon Redshift, and Amazon ES offer many instance types. In some cases, you should select the cheapest type that suits your workload's requirements. In other cases, using fewer instances of a larger instance type might result in lower total cost or better performance. You should benchmark your application environment and select the right instance type depending on how your workload uses CPU, RAM, network, storage size, and I/O.

Similarly, you can reduce cost by selecting the right storage solution for your needs. For example, Amazon S3 offers a variety of storage classes, including Standard, Reduced Redundancy, and Standard-Infrequent Access. Other services, such as Amazon EC2, Amazon RDS, and Amazon ES, support different EBS volume types (magnetic, general purpose SSD, provisioned IOPS SSD) that you should evaluate.

Over time, you can continue to reduce cost with continuous monitoring and tagging. Just like application development, cost optimization is an iterative process. Because, your application and its usage will evolve over time, and because AWS iterates frequently and regularly releases new options, it is important to continuously evaluate your solution.

AWS provides tools to help you identify those cost-saving opportunities and keep your resources right-sized.[48] To make those tools' outcomes easy to interpret, you should define and implement a tagging policy for your AWS resources. You can make tagging a part of your build process and automate it with AWS management tools such as AWS Elastic Beanstalk and AWS OpsWorks. You can also use the managed rules provided by AWS Config to assess whether specific tags are applied to your resources or not.

Elasticity

Another way you can save money with AWS is by taking advantage of the platform's elasticity. Plan to implement Auto Scaling for as many Amazon EC2 workloads as possible, so that you horizontally scale up when needed and scale down and automatically reduce your spending when you don't need that capacity anymore. In addition, you can automate turning off non-production workloads when not in use.[49] Ultimately, consider which compute workloads you could implement on AWS Lambda so that you never pay for idle or redundant resources.

Where possible, replace Amazon EC2 workloads with AWS managed services that either don't require you to make any capacity decisions (such as ELB, Amazon CloudFront, Amazon SQS, Amazon Kinesis Firehose, AWS Lambda, Amazon SES, Amazon CloudSearch, or Amazon EFS) or enable you to easily modify capacity as and when need (such as Amazon DynamoDB, Amazon RDS, or Amazon ES).

Take Advantage of the Variety of Purchasing Options

Amazon EC2 On-Demand instance pricing gives you maximum flexibility with no long-term commitments. Two other EC2 instances that can help you reduce spending are *Reserved Instances* and *Spot instances*.

Reserved Instances

Amazon EC2 Reserved Instances allow you to reserve Amazon EC2 computing capacity in exchange for a significantly discounted hourly rate compared to On-Demand instance pricing. This is ideal for applications with predictable minimum capacity requirements. You can take advantage of tools such as AWS Trusted Advisor or Amazon EC2 usage reports to identify the compute resources that you use most often and that you should consider reserving. Depending on your Reserved Instance purchases, the discounts will be reflected in the monthly bill. There is technically no difference between an On-Demand EC2 instance and a Reserved Instance. The difference lies in the way you pay for instances that you reserve.

Reserved capacity options exist for other services as well (e.g., Amazon Redshift, Amazon RDS, Amazon DynamoDB, and Amazon CloudFront).

Tip: You should not commit to Reserved Instance purchases before you have sufficiently benchmarked your application in production. After you have purchased reserved capacity, you can use the Reserved Instance utilization reports to make sure you are still making the most of your reserved capacity.

Spot Instances

For less steady workloads, consider using Spot instances. Amazon EC2 Spot instances allow you to use spare Amazon EC2 computing capacity. Since Spot instances are often available at a discount compared to On-Demand pricing, you can significantly reduce the cost of running your applications.

Spot instances enable you to request unused EC2 instances, which can lower your Amazon EC2 costs significantly. The hourly price for a Spot instance (of each instance type in each Availability Zone) is set by Amazon EC2, and adjusted gradually based on the long-term supply of, and demand for, Spot instances. Your Spot instance runs whenever capacity is available and the maximum price per hour for your request exceeds the Spot price.

As a result, Spot instances are great for workloads that can tolerate interruption. You can, however, also use Spot instances when you require more predictable availability. For example, you can combine Reserved, On-Demand, and Spot instances to combine a predictable minimum capacity with *opportunistic* access to additional compute resources, depending on the Spot market price. This is a great, cost-effective way to improve throughput or application performance.

Caching

Caching is a technique that stores previously calculated data for future use. This technique is used to improve application performance and increase the cost efficiency of an implementation. It can be applied at multiple layers of an IT architecture.

Application Data Caching

Applications can be designed so that they store and retrieve information from fast, managed, in-memory caches. Cached information might include the results of I/O-intensive database queries, or the outcome of computationally intensive processing. When the result set is not found in the cache, the application can calculate it, or retrieve it from a database or expensive, slowly mutating third-party content, and store it in the cache for subsequent requests. However, when a result set is found in the cache, the application can use that result directly, which improves latency for end users and reduces load on back-end systems. Your application can control how long each cached item remains valid. In some cases, even a few seconds of caching for very popular objects can result in a dramatic decrease on the load for your database.

Amazon ElastiCache is a web service that makes it easy to deploy, operate, and scale an in-memory cache in the cloud. It supports two open-source, in-memory caching engines: Memcached and Redis. For more details on how to select the right engine for your workload, as well as a description of common ElastiCache design patterns, see the Performance at Scale with Amazon ElastiCache whitepaper.[50]

Amazon DynamoDB Accelerator (DAX) is a fully managed, highly available, in-memory cache for DynamoDB that delivers performance improvements from milliseconds to microseconds, for high throughput. DAX adds in-memory acceleration to your DynamoDB tables without requiring you to manage cache invalidation, data population, or cluster management.

Edge Caching

Copies of static content (images, CSS files, or streaming pre-recorded video) and dynamic content (responsive HTML, live video) can be cached at an Amazon CloudFront edge location, which is a CDN with multiple points of presence around the world. Edge caching allows content to be served by infrastructure that is closer to viewers, which lowers latency and gives you the high, sustained

data transfer rates necessary to deliver large popular objects to end users at scale.

Requests for your content are intelligently routed to Amazon S3 or your origin servers. If the origin is running on AWS, requests are transferred over optimized network paths for a more reliable and consistent experience. You can use Amazon CloudFront to deliver your entire website, including non-cachable content. In this case, the benefit is that Amazon CloudFront reuses existing connections between the Amazon CloudFront edge and the origin server, which reduces connection setup latency for each origin request. Other connection optimizations are also applied to avoid internet bottlenecks and fully use available bandwidth between the edge location and the viewer. This means that Amazon CloudFront can expedite the delivery of your dynamic content and provide your viewers with a consistent and reliable, yet personalized, experience when navigating your web application. Amazon CloudFront also applies the same performance benefits to upload requests as those applied to the requests for downloading dynamic content.

Security

Most of the security tools and techniques that you might already be familiar with in a traditional IT infrastructure can be used in the cloud. At the same time, AWS allows you to improve your security in a variety of ways. AWS is a platform that allows you to formalize the design of security controls in the platform itself. It simplifies system use for administrators and your IT department, and makes your environment much easier to audit in a continuous manner. For a detailed view on how you can achieve a high level of security governance, see the Security at Scale: Governance in AWS[51] and the AWS Security Best Practices whitepapers.[52]

Use AWS Features for Defense in Depth

AWS provides many features that can help you build architectures that feature defense in depth methods. Starting at the network level, you can build a VPC topology that isolates parts of the infrastructure through the use of subnets, security groups, and routing controls. Services like AWS WAF, a web application firewall, can help protect your web applications from SQL injection and other vulnerabilities in your application code. For access control, you can use IAM to define a granular set of policies and assign them to users, groups, and AWS resources. Finally, the AWS Cloud offers many options to protect your

data, whether it is in transit or at rest with encryption.[53] For more information about all of the available AWS security features, see the AWS Cloud Security page on the AWS website.[54]

Share Security Responsibility with AWS

AWS operates under a shared security responsibility model: AWS is responsible for the security of the underlying cloud infrastructure and you are responsible for securing the workloads you deploy in AWS. This helps you to reduce the scope of your responsibility and focus on your core competencies through the use of AWS managed services. For example, when you use services such as Amazon RDS and Amazon ElastiCache, security patches are applied automatically to your configuration settings. This not only reduces operational overhead for your team, but it could also reduce your exposure to vulnerabilities.

Reduce Privileged Access

When your servers are programmable resources, you get many security benefits. The ability to change your servers whenever you need to enables you to eliminate the need for guest operating system access to production environments. If an instance experiences an issue, you can automatically or manually terminate and replace it. However, before you replace instances, you should collect and centrally store log data that can help you recreate issues in your development environment and deploy them as fixes through your continuous deployment process. This approach ensures that the log data assist with troubleshooting and raise awareness of security events. This is particularly important in an elastic compute environment where servers are temporary. You can use Amazon CloudWatch Logs to collect this information. Where you don't have direct access, you can implement services such as AWS Systems Manager [55] to take a unified view and automate actions on groups of resources. You can integrate these requests with your ticketing system, so that access requests are tracked and dynamically handled only after approval.

Another common security risk is the use of stored, long term credentials or service accounts. In a traditional environment, service accounts are often assigned long-term credentials that are stored in a configuration file. On AWS, you can instead use IAM roles to grant permissions to applications running on EC2 instances through the use of short-term credentials, which are automatically distributed and rotated. For mobile applications, you can use Amazon Cognito to allow client devices to access AWS resources through

temporary tokens with fine-grained permissions. As an AWS Management Console user, you can similarly provide federated access through temporary tokens instead of creating IAM users in your AWS account. Then, when an employee leaves your organization and is removed from your organization's identity directory, that employee also automatically loses access to your AWS accounts.

Security as Code

Traditional security frameworks, regulations, and organizational policies define security requirements related to items such as firewall rules, network access controls, internal/external subnets, and operating system hardening. You can implement these in an AWS environment as well, but you now have the opportunity to capture them all in a template that defines a *Golden Environment*. This template is used by AWS CloudFormation and deploys your resources in alignment with your security policy. You can reuse security best practices among multiple projects, as a part of your continuous integration pipeline. You can perform security testing as part of your release cycle, and automatically discover application gaps and drift from your security policy.

Additionally, for greater control and security, AWS CloudFormation templates can be imported as *products* into AWS Service Catalog.[56] This allows you to centrally manage your resources to support consistent governance, security, and compliance requirements, while enabling your users to quickly deploy only the approved IT services they need. You apply IAM permissions to control who can view and modify your products, and you define constraints to restrict the ways that specific AWS resources can be deployed for a product.

Real-Time Auditing

Testing and auditing your environment is key to moving fast while staying safe. Traditional approaches that involve periodic (and often manual or sample-based) checks are not sufficient, especially in agile environments where change is constant. On AWS, you can implement continuous monitoring and automation of controls to minimize exposure to security risks. Services such as AWS Config, Amazon Inspector, and AWS Trusted Advisor continually monitor for compliance or vulnerabilities, giving you a clear overview of which IT resources are in compliance, and which are not. With AWS Config rules you also know if a resource was out of compliance even for a brief period of time, making both point-in-time and period-in-time audits very effective. You can implement

extensive logging for your applications (using Amazon CloudWatch Logs) and for the actual AWS API calls by enabling AWS CloudTrail.[57]

AWS CloudTrail is a web service that records API calls to supported AWS services in your AWS account and delivers a log file to your S3 bucket. Log data can then be stored in an immutable manner and automatically processed to either send a notification or take an action on your behalf, protecting your organization from non-compliance. You can use AWS Lambda, Amazon EMR, Amazon ES, Amazon Athena, or third-party tools from AWS Marketplace to scan log data to detect events such as unused permissions, privileged account overuse, key usage, anomalous logins, policy violations, and system abuse.

Conclusion

When you design your architecture in the AWS Cloud, it is important to consider the important principles and design patterns available in AWS, including how to select the right database for your application, and how to architect applications that can scale horizontally and with high availability. Because each implementation is unique, you must evaluate how to apply this guidance to your implementation. The topic of cloud computing architectures is broad and continuously evolving. You can stay up-to-date with the latest changes and additions to the AWS cloud offerings with the material available on the AWS website[58] and the AWS training and certification offerings.[59]

Contributors

These individuals contributed to this document:

- Andreas Chatzakis, Manager, AWS Solutions Architecture
- Paul Armstrong, Principal Solutions Architect, AWS

Further Reading

For more architecture examples, see the AWS Architecture Center.[60]

For applications already running on AWS, we recommend you review the AWS Well-Architected Framework whitepaper, which provides a structured evaluation model.[61]

For information to help you validate your operational readiness, see the comprehensive AWS Operational Checklist.[62]

Document Revisions

Date	Description
October 2018	Document review and update
June 2013	First publication

Notes

[1] http://do.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf

[2] https://aws.amazon.com/about-aws/

[3] https://aws.amazon.com/about-aws/global-infrastructure/

[4] For example, see the PHP Amazon DynamoDB session handler (http://docs.aws.amazon.com/aws-sdk-php/v3/guide/service/dynamodb-session-handler.html) and the Tomcat Amazon DynamoDB session handler (http://docs.aws.amazon.com/AWSSdkDocsJava/latest//DeveloperGuide/java-dg-tomcat-session-manager.html)

[5] https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-target-groups.html#sticky-sessions

[6] https://do.awsstatic.com/whitepapers/Big_Data_Analytics_Options_on_AWS.pdf

[7] https://d1.awsstatic.com/whitepapers/core-tenets-of-iot1.pdf

[8] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html

[9] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-custom-resources-lambda.html

[10] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

[11] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/concepts.platforms.html

[12] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/ebextensions.html

[13] https://do.awsstatic.com/whitepapers/overview-of-deployment-options-on-aws.pdf

[14] https://do.awsstatic.com/whitepapers/managing-your-aws-infrastructure-at-scale.pdf

[15] https://do.awsstatic.com/whitepapers/DevOps/infrastructure-as-code.pdf

[16] https://docs.aws.amazon.com/lambda/latest/dg/automating-deployment.html

[17] https://aws.amazon.com/elasticbeanstalk/

[18] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-recover.html

[19] https://aws.amazon.com/ec2/systems-manager/

[20] https://aws.amazon.com/autoscaling/

[21] http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/AlarmThatSendsEmail.html

[22] http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/WhatIsCloudWatchEvents.html

[23] http://docs.aws.amazon.com/lambda/latest/dg/with-scheduled-events.html

[24] https://aws.amazon.com/answers/security/aws-waf-security-automations/

[25] https://www.awsarchitectureblog.com/2015/03/backoff.html

[26] http://aws.amazon.com/products/

[28] https://d0.awsstatic.com/whitepapers/AWS_Serverless_Multi-Tier_Architectures.pdf

[29] http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html

[30] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_BestPractices.html

[31] https://aws.amazon.com/nosql/

[32] https://aws.amazon.com/dynamodb/dax/

[33] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BestPractices.html

[34] https://d0.awsstatic.com/whitepapers/migration-best-practices-rdbms-to-dynamodb.pdf

[35] https://aws.amazon.com/redshift/faqs/

[36] https://aws.amazon.com/documentation/cloudsearch/

[37] https://aws.amazon.com/documentation/elasticsearch-service/

[38] https://aws.amazon.com/neptune/

[39] https://d0.awsstatic.com/whitepapers/Storage/data-lake-on-aws.pdf

[40] https://d0.awsstatic.com/whitepapers/aws-building-fault-tolerant-applications.pdf

[41] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-recover.html

[42] http://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html

[43] http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html

[44] https://media.amazonwebservices.com/AWS_Disaster_Recovery.pdf

[45] https://aws.amazon.com/architecture/well-architected/

[46] http://www.awsarchitectureblog.com/2014/04/shuffle-sharding.html

[47] https://d0.awsstatic.com/whitepapers/Cost_Optimization_with_AWS.pdf

[48] http://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/monitoring-costs.html

[49] http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/UsingAlarmActions.html

[50] https://d0.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticache.pdf

[51] https://d0.awsstatic.com/whitepapers/compliance/AWS_Security_at_Scale_Governance_in_AWS_Whitepaper.pdf

[52] https://d0.awsstatic.com/whitepapers/aws-security-best-practices.pdf

[53] https://d0.awsstatic.com/whitepapers/aws-security-best-practices.pdf

[54] http://aws.amazon.com/security

[55] https://aws.amazon.com/systems-manager/

[56] https://aws.amazon.com/servicecatalog/

[57] https://d0.awsstatic.com/whitepapers/compliance/AWS_Security_at_Scale_Logging_in_AWS_Whitepaper.pdf

[58] https://aws.amazon.com/

[59] https://aws.amazon.com/training/

[60] https://aws.amazon.com/architecture

[61] http://do.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf

[62] https://do.awsstatic.com/whitepapers/aws-operational-checklists.pdf