

Modern Application Development on AWS

Cloud-Native Modern Application Development and
Design Patterns on AWS

October 2019



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.

© 2019 Amazon Web Services, Inc. or its affiliates. All rights reserved.

Contents

Introduction	6
Accelerating the Innovation Flywheel	6
Modern Application Development	7
Capabilities of Modern Applications	7
Best Practices of Modern Application Development	9
Modern Application Design Patterns	14
Implementing Microservice Architectures using AWS Services	14
Continuous Integration and Continuous Delivery on AWS	30
CI/CD Services on AWS	30
CI/CD Patterns for Different Application Types	33
Conclusion	38
Contributors	38
Further Reading	39
AWS Services	39
Whitepapers	40
Video	40
Document Revisions	40

Abstract

Modern application development using containers and serverless technologies can help your organization accelerate innovation. This paper includes information about important best practices and design patterns that you can use to build your modern application in the AWS Cloud.

Introduction

Modern companies are increasingly global, and their products are increasingly digital. These digital products—such as cloud infrastructure, mobile apps, big data pipelines, and social media—are influencing application development, which requires an unprecedented pace of change for companies. To achieve this speed, business leaders must adapt their culture, processes, and technologies to the new reality of this digital age.

Rapid innovation is vital for modern companies, which must drive growth by making the most of their human resources, seeking out new opportunities, and nurturing new ideas. Digital technology is at the core of this rapid innovation.

Accelerating the Innovation Flywheel

Businesses in almost all industries are experiencing an unprecedented pace of change, and rapid innovation is crucial to improving their pace. Small, unknown competitors can get ahead in a matter of months by focusing on innovation, so it is essential to not only innovate, but to do so quickly.

Amazon has learned that experiments let you innovate faster. To accelerate innovation, we perform an experiment, listen to user feedback, and experiment again. We do not fear failure, but apply the learnings from each experiment in future efforts. We call this the *innovation flywheel*. To spin this flywheel rapidly, we need a system to release products, collect feedback, add new features, and release again. The features of modern applications make this process possible, and enable you to spin the flywheel and get ahead of the competition through rapid innovation.

Modern Application Development

The most successful companies recognize that it is their technology that sets them apart from the competition. To keep growing and winning business, companies need to invent new products rapidly. To promote a culture of innovation that makes this possible, successful companies continually update their methods of designing, building, and administering applications. We call this *modern application development*.

Modern application development gives companies a competitive edge by enabling them to innovate more rapidly. Companies that embrace innovation can complete more experiments and bring ideas to market more quickly by shifting resources from undifferentiated heavy lifting—such as administering and provisioning infrastructure—to more valuable activities.

Modern application development practices can help companies to realize the speed and agility that go with innovation. Some customers take their on-premises virtual machines (VMs) and move them (also known as lift-and-shift) to host them on Amazon Elastic Compute Cloud (Amazon EC2¹). Other customers change the platform of their applications to a container-based model that is more optimized for the cloud. Still other companies refactor their monolithic applications and transition to a microservice-based architecture. Most companies find that when they build more cloud-native applications, they spend less time on administrative overhead and can focus more on their core business.

Capabilities of Modern Applications

Modern applications should be:

- **Secure** – It is crucial for any application to be secure. Security measures must be implemented not only in a certain piece of the application, but in all layers and at each stage of the lifecycle.
- **Resilient** – A modern application is resilient. For example, if an application encounters a failure when it calls an external data source, it should retry or otherwise handle the exception—not become unresponsive—while continuing to operate with a graceful degradation of functionality. This pattern also applies to a microservice architecture² and interactions with other services.

- **Elastic** – By flexibly scaling out and scaling in depending on the rate of requests or other metrics, modern applications can optimize cost without missing business opportunities. Automating the process of scaling out and scaling in, or using managed services that include auto scaling functionality, reduces routine administrative burden and prevents the extreme disruption of outages.
- **Modular** – Modern applications are modular, with high cohesion and loose coupling. Larger systems should not be single monoliths, but should be separated along domain boundaries into different components, each with a distinct responsibility. Not only does this separation allow for greater availability and scalability, but frequent releases are easier, because different components can be deployed independently.
- **Automated** – Integration and deployment of modern applications must be automated to enable frequent, high-quality releases. In addition to being error prone, manual processes can introduce dependence on individual people, such as requiring a single administrator to make deployments. To support agile development and frequent releases, modern applications should be deployed through continuous integration and continuous delivery (CI/CD) pipelines. In a CI/CD model, code is pushed to version control, tests are run in a clean CI environment, and deployments are performed automatically if all tests pass.
- **Interoperable** – In modern applications, each service must interact with other services, provide the resources requested of it, and perform the tasks expected of it. It must be possible to add functionality to different services independently and continue to release frequently, without impacting other services. This means that services must keep their implementation details private, exposing all required functionality through robust, public APIs. These public APIs must also be stable and backward compatible to allow for independent releases.

There are various methods you can use to implement modern applications. This paper includes information about methods to deploy applications in the cloud with containers and serverless technology.

Best Practices of Modern Application Development

Through conversations with customers and our own development teams, we found that there are several modern application development best practices shared by organizations that bring innovative ideas to the market rapidly.

Security and Compliance

When you build your system in the AWS Cloud, we recommend that you always start with security and compliance. Securing the whole application lifecycle enables organizations to address security threats without sacrificing speed of innovation. For example:

- **Authentication** – Control access to your system with permission settings that prevent malicious access. AWS administrators can sign in to the AWS Console with AWS Identity and Access Management (IAM) credentials, or through integrations with Microsoft Active Directory or a SAML Identity Provider. Applications built on AWS can leverage Amazon Cognito to allow end users to authenticate and access resources.
- **Authorization** – Implement role-based access control with flexible policies that restrict the use of resources without overly complicated administration. IAM provides granular authorization policies for any AWS resources.
- **Auditing and Governance** – Evaluate the behavior of workloads and make sure that they conform to compliance requirements and your organization's standards. AWS CloudTrail can audit interactions with AWS APIs and log aggregation with Amazon CloudWatch enables you to audit your applications. AWS Config can make sure that AWS resources are configured to align with your organization's standards.
- **Validation** – Test all aspects of application functionality, and make sure that it works as intended. Automate validation as much as possible with continuous integration and continuous delivery (CI/CD).

Modern applications should be thoroughly and frequently tested, however, this must not reduce development velocity. Similarly, you should limit developer permissions, but you should not revoke the access that they require. Build your security into the entire application lifecycle, and automate and continuously reevaluate your security processes and standards.

Microservice Architecture

As monolithic applications grow, it becomes difficult to modify or add functionality to them, and to track what parts of the codebase are involved in a specific change. As a result, small changes can require lengthy regression testing, and development of new features can slow. In an application built with a microservice architecture and loosely coupled components, many new features and bug fixes can be implemented at the level of a single service and released much more rapidly.

Organizations with monolithic legacy applications can become more agile and flexible by redesigning their applications into microservices. Each service is deployed separately, and all the services work together to offer the same functionality as the monolithic system. Microservices can be built, modified, and released quickly, which provides faster experimentation and innovation. Each team that builds microservices can also take clear ownership of their own design, development, deployment, and operations.

To achieve this loose coupling, the microservices in a system must communicate with each other. A datastore that is shared between services creates tight coupling, hidden dependencies, timing issues, and challenges with scaling and availability. It is better to use published APIs or asynchronous message queues to communicate between separate services. Separating processes into different pieces that are connected by messages in queues creates clear transaction boundaries and enables services to operate more independently.

Messaging systems can provide scalability, resilience, availability, consistency, and distributed transactions because of the following characteristics:

- Trusted and resilient message delivery system
- Non-blocking and one-way operation
- Loosely coupled services
- Bringing focus to different logical components in the system, and allowing each to work independently

Architectures that take advantage of these elements can easily expose robust APIs and asynchronous communication channels, which enables each service to be operated and automated independently, and which also improves reliability.

When many different microservices are connected to perform a process, you must have a method to monitor the state of a single end-to-end task. You must also make sure that

all the necessary steps happen in the correct order and at the correct time. You can use state machines to both monitor the state of tasks and make sure they occur in the correct order.

You also need a method to manage the overall workflow between services, to configure various timeouts, cancellations, heartbeats for long-running tasks, and granular monitoring and auditing. Managing services with this type of tooling improves speed, productivity, and flexibility. To make sure that microservices execute in the correct sequence with appropriate timing, modern applications take advantage of orchestration and messaging tools. Using orchestration tools makes it easy to build robust services in a repeatable way. AWS Step Functions is a fully managed tool that can coordinate arbitrary workflows across services. When you use messaging tools, you remove direct dependencies between services, which improves reliability and scalability. You can use different tools—such as Amazon Simple Queue Service (Amazon SQS), Amazon CloudWatch Events, and Amazon Kinesis—depending on the specific workload. By using orchestration and messaging tools together, your developers do not have to spend time on workflow execution, state management, and inter-service communication, which gives them valuable time to focus on core business logic.

Using Serverless Technology

When you operate and maintain the servers and operating systems (OS) that run your organization's applications, your system administrators must spend time completing simple and repetitive tasks, such as applying OS security patches. Instead of scaling up by request volume, they must provision servers for peak volume ahead of time, while carefully considering availability and durability requirements. You might also have to pay for all of this overprovisioned infrastructure in advance, instead of paying for what you use as you go.

Though services such as AWS Auto Scaling and AWS Systems Manager can reduce these burdens on conventional, VM-based infrastructure, when you build your system on serverless technology, you don't have to provision and manage servers. Your administrators don't have to spend time on OS patches, or maintain unused resources to be prepared for occasional peak usage. Serverless applications scale to meet the precise demand on each component. Reliability and fault-tolerance are also largely built-in by default, which eliminates much of the design and operations time required for these aspects of the system. By building modern applications with serverless technologies from the beginning, the whole lifecycle of building, deploying, and running applications can also be kept secure. When you remove operational complexity, your

developers can focus their time and energy on building products that delight your customers.

AWS provides serverless computing services such as AWS Lambda³ and AWS Fargate⁴. There is Amazon Simple Storage Service (Amazon S3)⁵ for object storage, and there are now two serverless database options: Amazon DynamoDB⁶, a fast and flexible NoSQL database, and Amazon Aurora Serverless⁷, an on-demand and auto-scaling configuration for Amazon Aurora. If you want to build an end-to-end serverless application, compute, database, and storage services might not be enough. You can use other serverless AWS offerings⁸ throughout your workload, from API management, messaging, and orchestration, to troubleshooting and monitoring.

Automating Deployment with CI/CD

Companies strive to innovate quickly to deliver the most value they can to customers as quickly as possible. To achieve this, modern applications use continuous integration and continuous delivery (CI/CD) to automate the entire release process: building and running tests, promoting artifacts to staging, and the final deployment to production. CI/CD can also automate certain security controls, such as scanning for known vulnerabilities and performing static analysis. The full CI/CD pipeline can consist of an arbitrary number of quality gates and controls, all of which must be passed successfully before any new code makes it to production.

By automating the full build/test/deploy process, it becomes not only more reproducible, but faster as well. It can also be performed much more frequently—perhaps many times a day—meaning that each individual deployment consists of fewer changes and less risk. Instead of being a high-risk, all-hands-on-deck event, CI/CD allows deployments to production to be mundane affairs. Finally, because the time from when code is committed to when it is deployed is so much shorter than with manual processes, high-priority security fixes or config changes no longer require special hot patches, but can flow through the standard pipeline.

AWS customers can take advantage of fully-managed CI/CD services such as AWS CodeBuild, AWS CodePipeline, and AWS CodeDeploy, in addition to open-source options and third-party marketplace offerings.

Managing Infrastructure as Code

To get the full benefits of CI/CD, you should create a model for your entire application and infrastructure as code (IaC). By modeling infrastructure as code, you can incorporate it into your standard application development lifecycle, execute infrastructure changes in your CI/CD pipeline, and get additional benefits, such as reducing configuration errors and provisioning faster. AWS provides a number of IaC tools. One tool is AWS CloudFormation⁹, which is a service that lets you specify any cloud infrastructure you need in a simple template file, and then provisions the infrastructure for you. Another tool is AWS Serverless Application Model (SAM)¹⁰, which builds on AWS CloudFormation with additional tooling and convenience functions for building serverless applications. AWS Cloud Development Kit (CDK)¹¹ is a tool that provides a framework to design cloud infrastructure in code using a language of your choice and then provisions it with CloudFormation.

Monitoring and Logging

Developers of modern applications should monitor the behavior of their application at runtime using monitoring and logging tools, and use that data to maintain or improve their customers' experience. In modern digital products, this could mean monitoring a many data types, including application logs, data from mobile devices, web click streams, IoT sensor data, or other usage data. Modern application developers should take advantage of all of this data as they continue to expand and enhance their products.

On AWS, you can set up monitoring, logging, and alarms for all your application components using Amazon CloudWatch. For more information on logging, see [Log Aggregation](#).

Modern Application Checklist

Use the following information to verify the modernization level of your application:

- Security and compliance are built in throughout the application lifecycle
- Application is structured as a collection of microservices
- Serverless technologies are used wherever possible
- CI/CD is used to deliver high-quality functionality quickly
- Infrastructure is developed and deployed as code
- Monitoring tools are used to gain insight into the behavior of the application

Modern Application Design Patterns

A best practice for modern application development is to use patterns to design and implement your applications. Using AWS services as building blocks for these applications, you can greatly reduce your implementation effort and achieve reliability and availability, which enables your developers to focus on business logic that adds value to your applications.

Implementing Microservice Architectures using AWS Services

You can use common patterns for microservices, following best practices, and implement them using AWS services.

API Gateways

The API gateway pattern can be used when there are many calls to backend services, and when the content provided varies depending on the client interface or device type. API gateways can consolidate different backend services behind a unified API and serve the content required for each device.

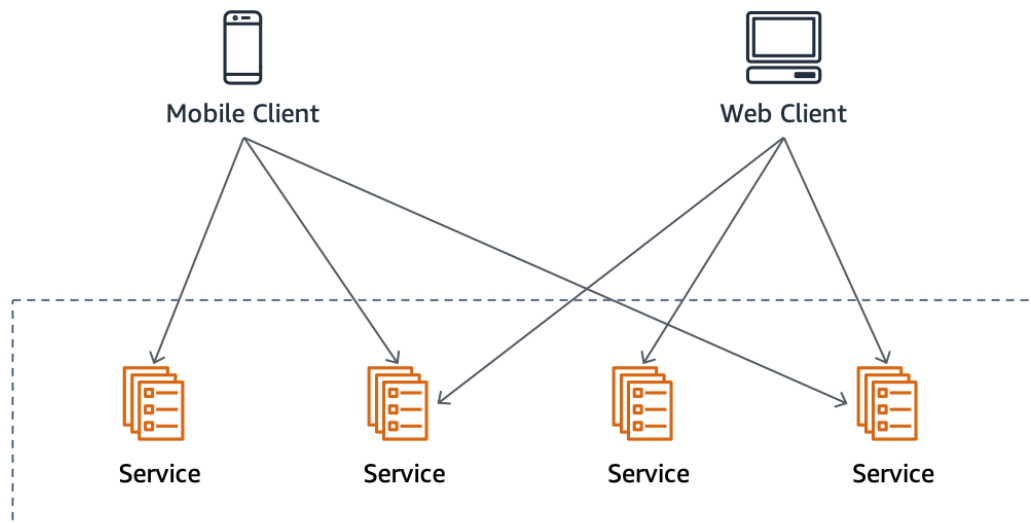


Figure 1 – Example of communication between services and mobile devices and computer browsers without an API gateway

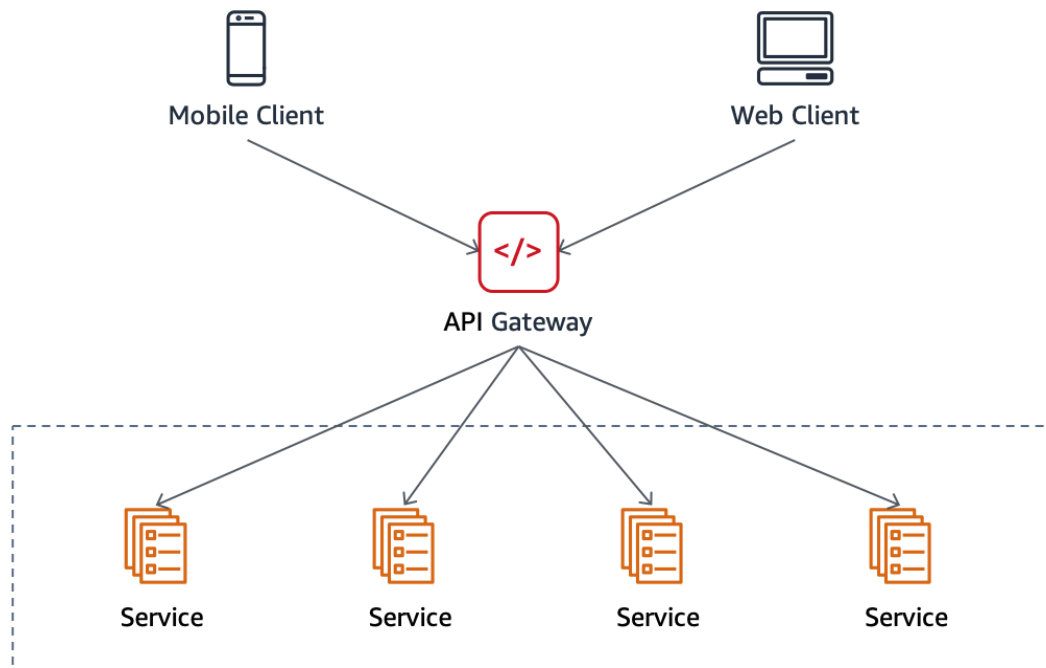


Figure 2 – Example of communication between services and mobile devices and computer browsers with an API gateway

If you plan to use the API gateway pattern in the AWS Cloud, you can use Amazon API Gateway¹² to integrate with backend endpoints. Amazon API Gateway also enables you to create, publish, maintain, monitor, and protect REST or WebSocket APIs at any scale.

Amazon API Gateway provides many other capabilities required of production-grade APIs, such as throttling, caching, logging, API tokens, authentication or authorization integrated with Amazon Cognito, custom authorizers, and proxying of requests to other AWS services. One essential AWS service that Amazon API Gateway can send proxy requests to is AWS Lambda, which is the foundation for creating arbitrary web services without managing any server infrastructure.

Because Amazon API Gateway is managed by AWS, you don't have to worry about operating and maintaining it. Using Amazon API Gateway provides improved security, reliability, and availability, which allows your developers to spend more time on core application functionality.

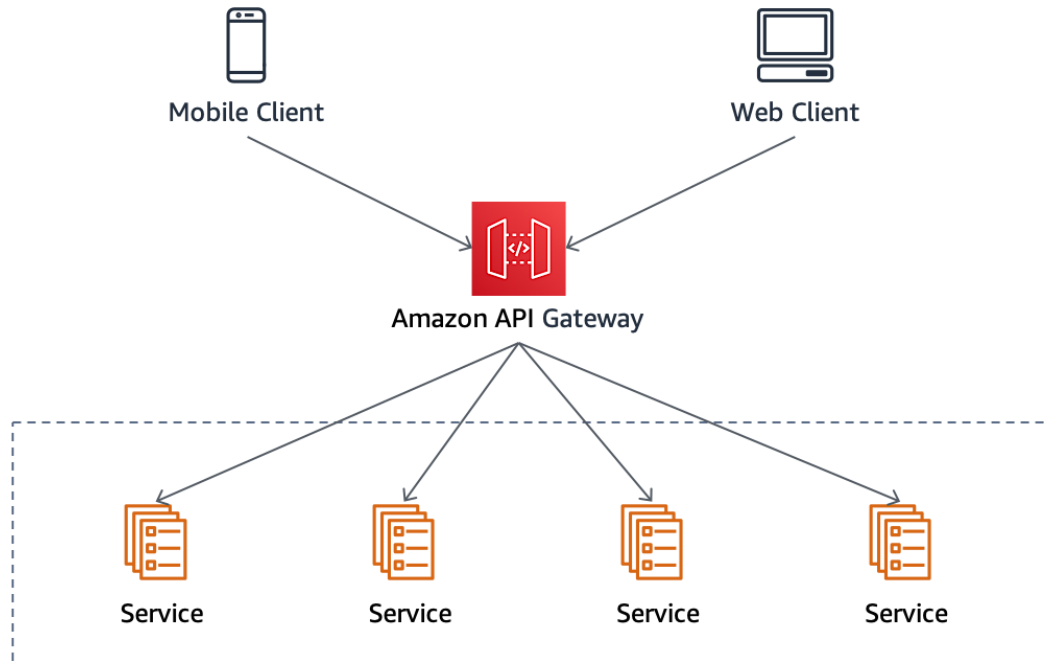


Figure 3 – Example of communication between services and mobile devices and computer browsers with Amazon API gateway

Service Discovery and Service Registries

When a system includes multiple microservices, services must be able to find the location of the other services that they depend on. Microservices must be scalable and elastic, and if components fail, new instances or containers must be brought online to ensure constant availability. This means that the IP addresses of the instances or containers in a microservice can be constantly changing. Each instance of a service also must be continually monitored for availability. You can use load balancers to provide stable, available endpoints, which are usually the best choice for public-facing web endpoints. However, load balancers require additional compute resources and introduce latency. If the client is under your control, as are the calls between microservices, it can be more efficient to use a *service discovery* pattern, which you can also think of as client-side load balancing.

In the service discovery pattern, information about the services to be discovered must be registered somewhere. A *service registry* is a central location where services to be called can store information about themselves as each individual container or instance starts up.

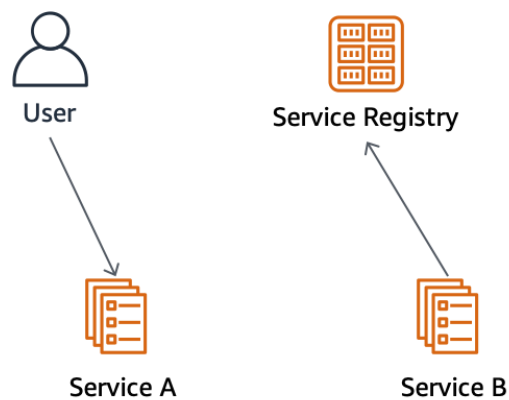


Figure 4 – Example of a service registry pattern

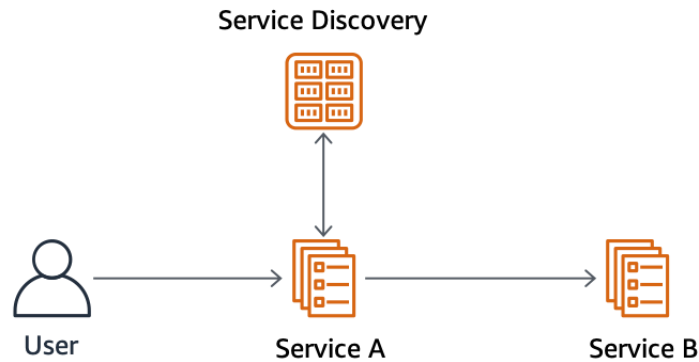


Figure 5 – Example of a service discovery pattern

You can use AWS Cloud Map to implement a service registry and service discovery pattern in the AWS Cloud. AWS Cloud Map is a fully managed service that allows clients to look up IP address and port combinations of service instances using DNS, and to dynamically retrieve abstract endpoints, such as URLs or Amazon Resource Names (ARNs) over the HTTP-based service discovery API.

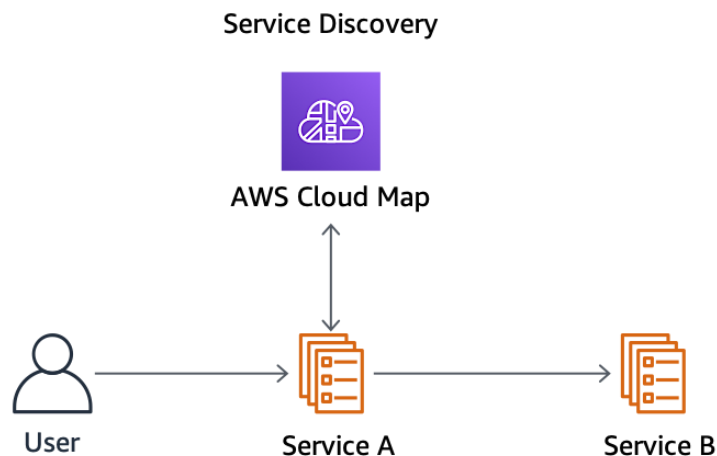


Figure 6 – Example of a service registry and service discovery pattern using AWS Cloud Map

Circuit Breaker

The *circuit breaker* pattern regulates the calls between microservices in your application. To respond to user requests, the microservices in your application make calls to each other. If Service A sends a call to Service B, but the return call from Service B is delayed or produces an error, then Service A returns an error to the user. If Service A retries the call instead of returning an error, it might provide a better user experience, but retries can produce extra load and long delays, and can end with an

error returned to the user. Instead, Service A should recognize that Service B is down, and degrade gracefully, if possible.

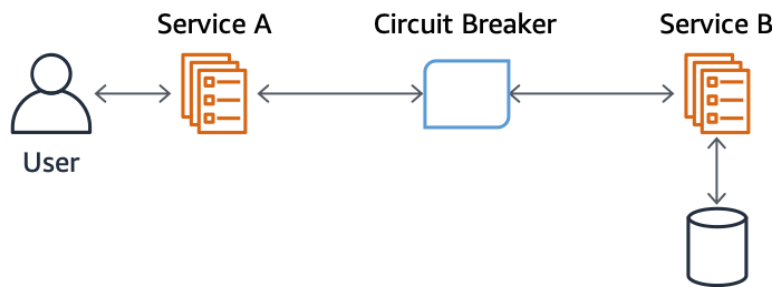


Figure 7 – Example of a circuit breaker pattern with returned calls between microservices

In the circuit breaker pattern, when calls to other services take longer than expected or return errors, the circuit breaker keeps count of the incidences and changes to the *open* state if the count exceeds the limit you configure. When in the open state, the circuit breaker returns errors to the caller immediately, without calling downstream services. After a fixed amount of time has passed, the circuit breaker returns to a *closed* state, which allows calls to the downstream service to return to normal.

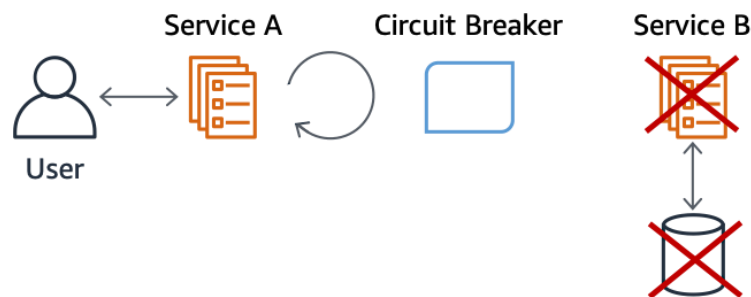


Figure 8 – Example of a circuit breaker pattern with errors returned immediately to the user

It was previously a best practice to implement circuit breakers using a library or framework in the service code, but now it is often handled in containerized microservices with *sidecars*. A sidecar is a separate helper container that is launched with the main container that exposes a core service. Envoy Proxy¹³ is one popular example of a sidecar. Though Envoy Proxy can be deployed on its own, it is often deployed as part of a service mesh. In this type of deployment, Envoy Proxy is the *data plane* and a tool such as AWS App Mesh or Istio is the *control plane*.

Command-Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) involves separating the data mutation or *command* part of a system from the *query* part. Updates and queries are conventionally completed using a single datastore. You can use CQRS to separate these two workloads if they have different requirements for throughput, latency, or consistency. When you separate command and query functions, you can scale them independently. For example, you can send queries to horizontally-scalable read replicas. For greater separation of command and query functions, you can use different data models and datastores for updates and queries. You can perform writes on a normalized model in a relational database through an ORM (object-relational mapping) and perform queries against a denormalized database that stores data in the same format required by an API (such as data transfer objects or *DTOs*), which reduces processing overhead.

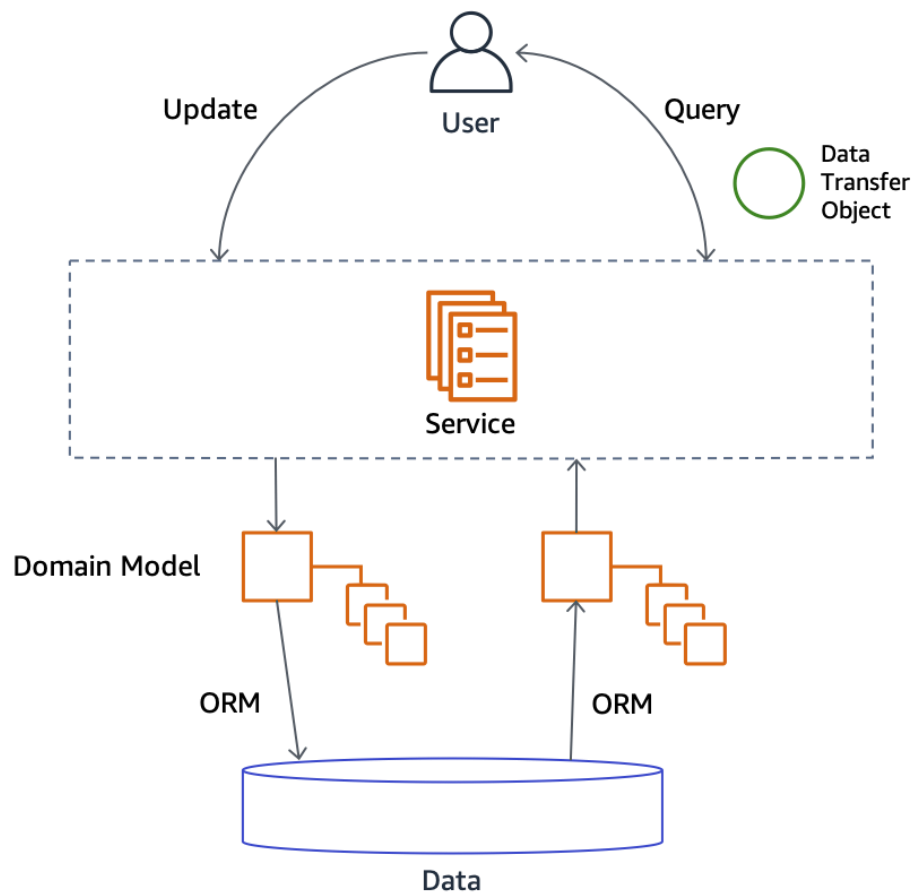


Figure 9 – Example of an architecture with updates and queries using a single datastore and ORM

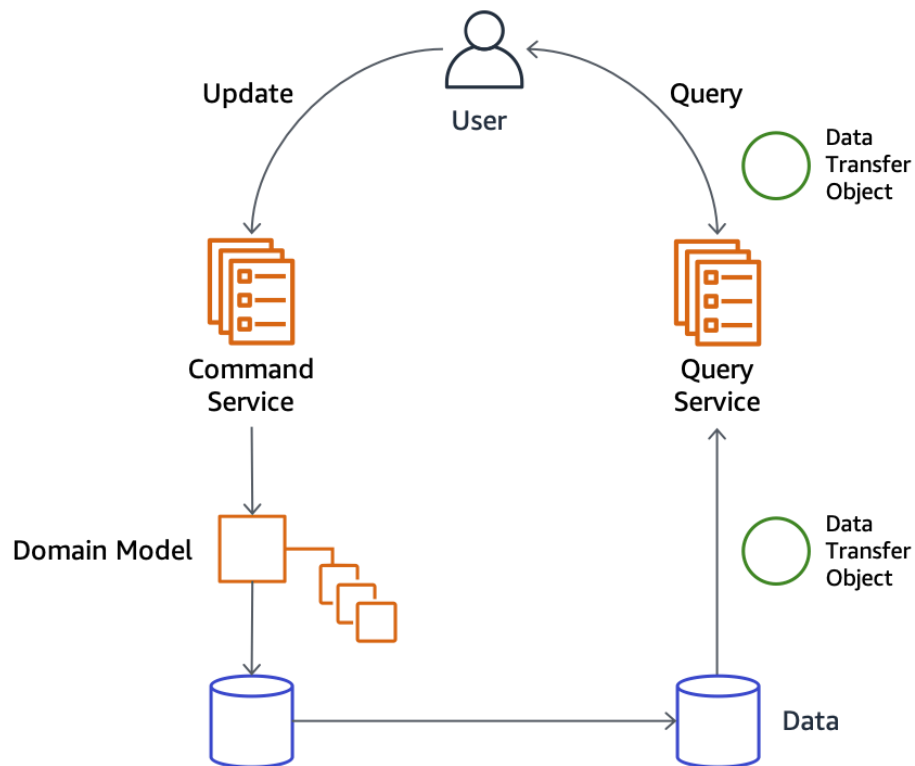


Figure 10 – Example of a CQRS architecture with separate command and query workloads and two datastores

Though this example optimizes your architecture for consistent writes in a relational database and very low-latency reads, you might instead want to optimize for very high write throughput and flexible query capabilities. In this situation, you can use a NoSQL datastore, such as Amazon DynamoDB, to get high write scalability on a workload with certain, well-defined access patterns when you add data. You can then use a relational database, such as Amazon Aurora, to provide complex, one-time query functionality. With this option, you can use Amazon DynamoDB streams that send data to an AWS Lambda function that makes appropriate updates to keep the data on Amazon Aurora up-to-date.

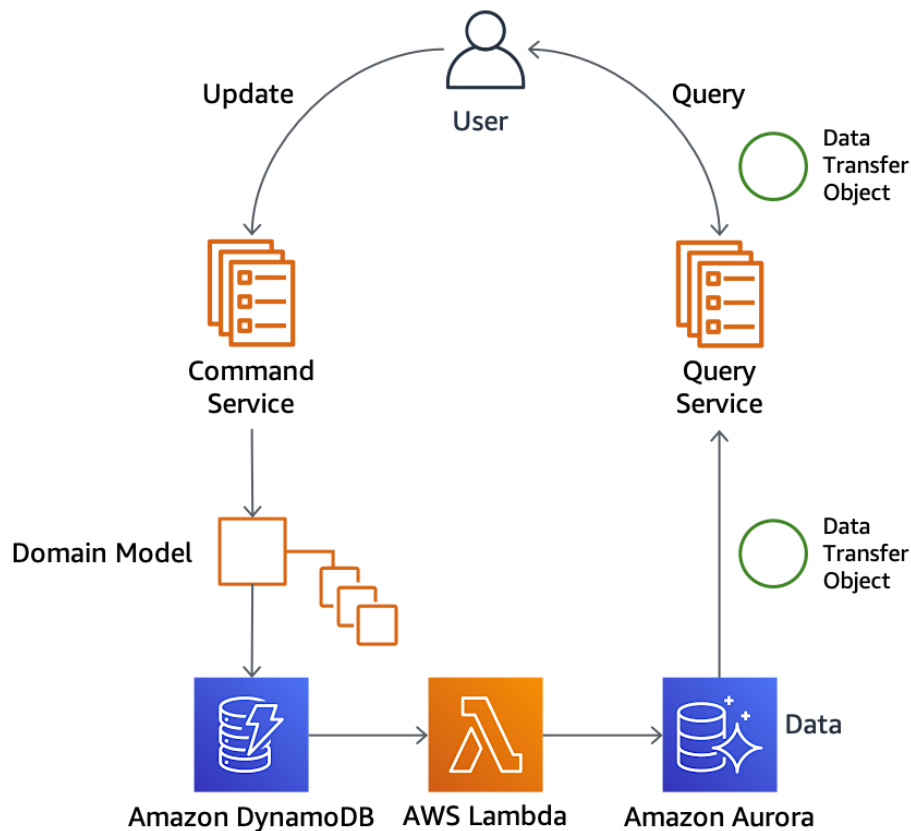


Figure 11 – Example of a CQRS architecture on AWS with DynamoDB, Lambda, and Aurora

You can also combine the *command* part of a CQRS architecture with the *event sourcing* pattern (see the following section). When you combine these patterns, you can rebuild the service query data model with the latest application state by replaying the update events. It is important to remember that the CQRS pattern generally results in eventual consistency between the queried datastore and the datastore that is written to.

Event Sourcing

With the *event sourcing* pattern, instead of updating data stores directly, any events with significance to business logic—such as orders being placed, credit inquiries being made, or orders being processed or shipped—are added to a durable event log. Because each event record is stored individually, all updates are *atomic* (indivisible and irreducible).

A key characteristic of this pattern is that the application state at any point in time can be rebuilt by simply reprocessing the stored events. Because data is stored as a series of events rather than through direct updates to data stores, various services can replay events from the event store to compute the appropriate state of their respective data stores. This works well with the CQRS pattern discussed previously, especially because

you can reproduce data for an event regardless of whether the command and query data stores have different schemas.

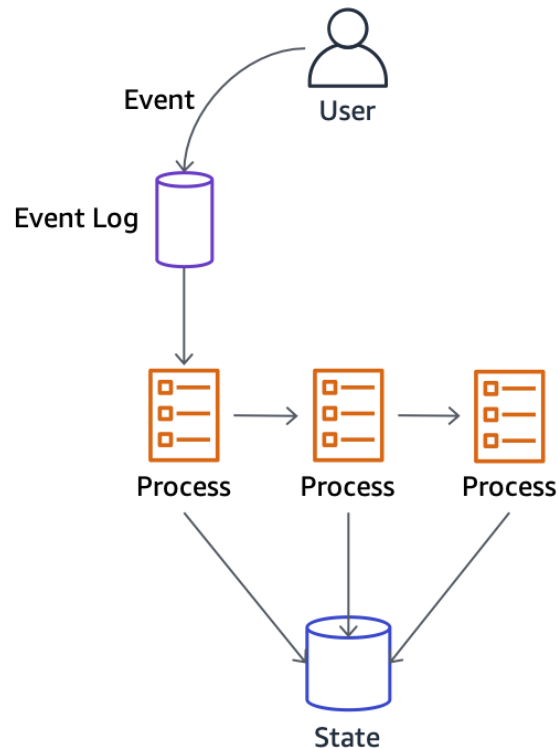


Figure 12 – Example of the event sourcing pattern

Because the event sourcing pattern involves storing and later replaying event messages, it requires some mechanism for storing and retrieving messages. If you plan to use this pattern in the AWS Cloud, depending on your use case, you can use Amazon Kinesis Data Streams¹⁴, Amazon Simple Queue Service (SQS)¹⁵, Amazon MQ¹⁶, or Amazon Managed Streaming for Kafka (Amazon MSK)¹⁷. In the event sourcing pattern, each event that changes the system is stored first to a message queue, and then updates to the application state are made based on that event. For example, an event can be written as a record in an Amazon Kinesis stream, and then a service built on AWS Lambda can retrieve the record and perform updates in its own data store).

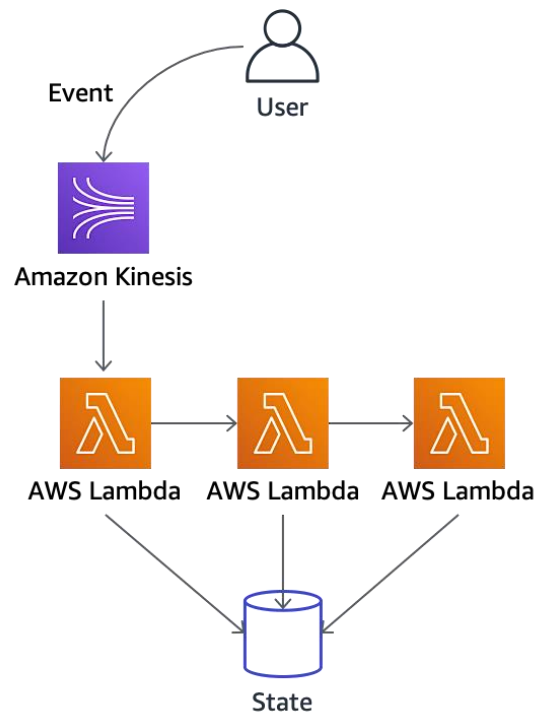


Figure 13 – Example of an event sourcing pattern using Amazon Kinesis and AWS Lambda

Sometimes it is useful to expand from one source of events to multiple targets. You can do this directly with Amazon Kinesis Data Streams, which allows multiple consumers to retrieve data from a stream. You can also use Amazon Simple Notification Service (Amazon SNS) to expand to multiple Lambda functions, which all listen to the same topic, and can propagate event data from Kinesis to other stateful components. With this configuration, you can also add an Amazon Simple Queue Service (Amazon SQS) queue between Amazon SNS and a given Lambda function that enables you to specify what causes the Lambda functions to execute.

Choreography

When a new customer creates an account on your website, they might need to save their profile information, receive a welcome email, and get credited with some initial points to use on the site. All of these activities are implemented by different services.

There are two implementation methods that you can consider to execute these tasks between your microservices: the *orchestration* pattern and the *choreography* pattern. With the *orchestration* pattern, similar to the relationship in a symphony between a conductor and an orchestra, there is a central service that issues commands to other services and makes sure that the entire process is completed. With the choreography pattern, just as dancers move independently after they have learned the choreography

of their dance, each service can execute independently in response to a particular event.

When you use the choreography pattern, an initial event that contains all the required information can be saved in a single message, and concludes an initial transaction. Other services can then retrieve that message asynchronously and complete their respective tasks. With this architecture, services are loosely coupled and do not have a direct impact on each other. The asynchronous relationship between saving and retrieving messages also provides scalability and reliability benefits.

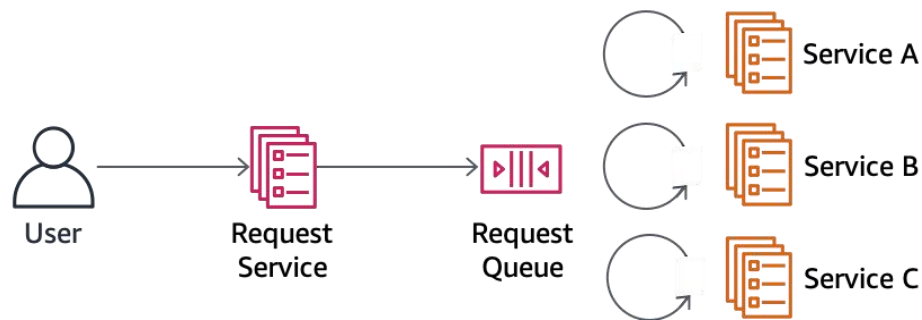


Figure 14 – Example of the choreography pattern

To implement the choreography pattern in the AWS Cloud, you can use Amazon Kinesis and AWS Lambda, or, depending on your requirements, use a combination of Amazon Simple Notification Service (Amazon SNS), Amazon Simple Queue Service (Amazon SQS), and AWS Lambda.

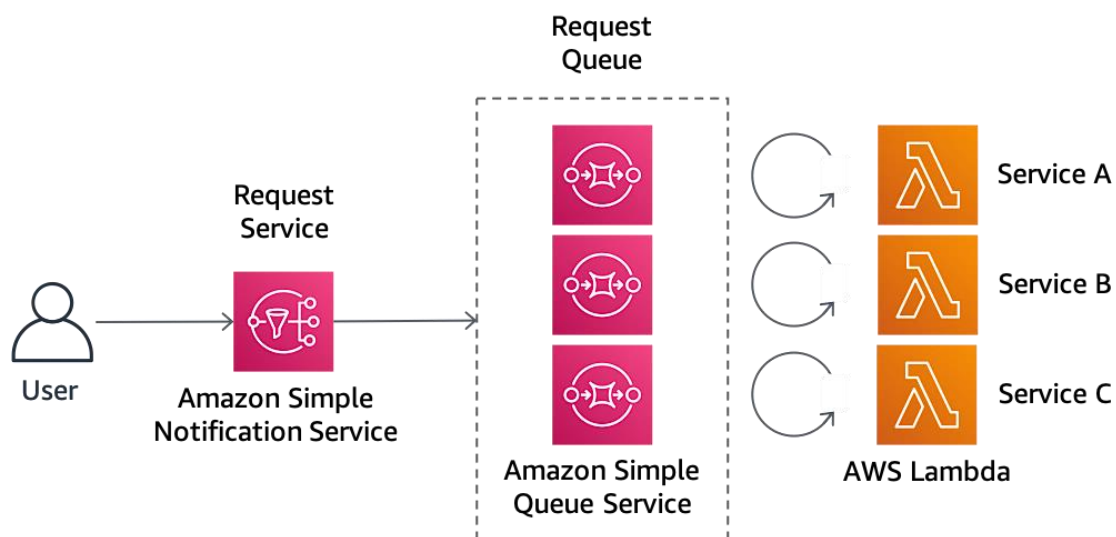


Figure 15 – Example of the choreography pattern using Amazon SNS, Amazon SQS, and AWS Lambda

Log Aggregation

The more complicated a system is, the more important it is to have good logs. The challenge is that if logs are scattered across different services, it's difficult to get a unified view of the entire system. It is essential to have a centralized place where logs are uniformly managed and discoverable. Gathering metrics is also important. In a microservice architecture, calls to various services might be required to handle a given request, so it can be more difficult to find the source of poor performance or errors compared to a monolith. This is why it's critical to have centrally aggregated logs and runtime metrics.

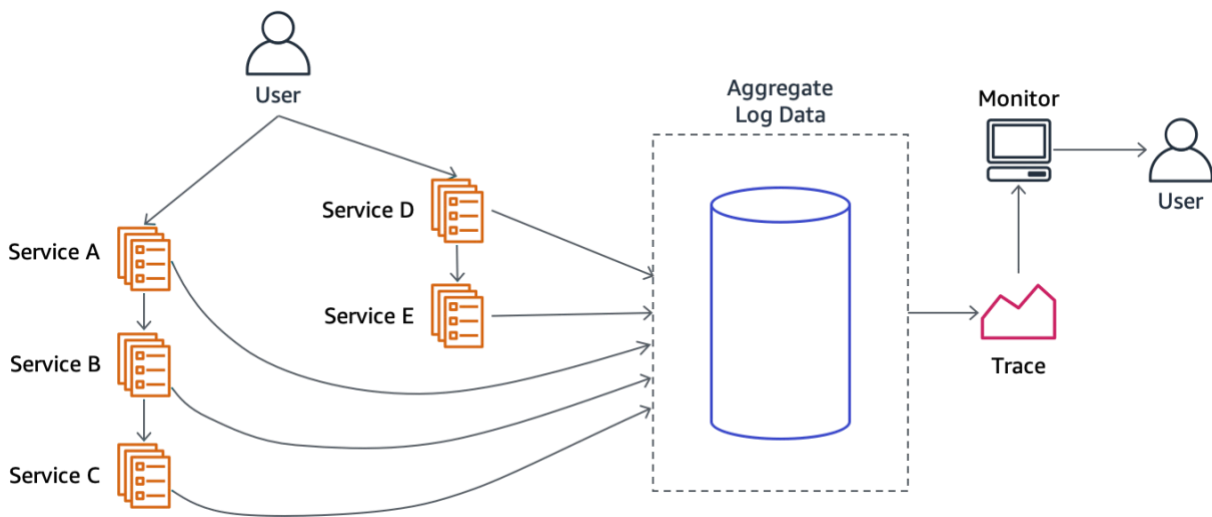


Figure 16 – Example of an architecture with log aggregation

For aggregated logging in the AWS Cloud, you can use Amazon CloudWatch Logs¹⁸. If you use AWS Lambda to implement microservices, anything you write to `stdout` is sent to CloudWatch Logs. Amazon Elastic Container Service (Amazon ECS) and AWS Fargate can also send anything written to `stdout` to Amazon CloudWatch Logs with the `awslogs` log driver. If you use Amazon Elastic Kubernetes Service (Amazon EKS), the logs can be sent to Amazon CloudWatch Logs using the sidecar pattern with Fluentd¹⁹ or Fluent Bit²⁰. CloudWatch Container Insights²¹ can also be used to send logs and metrics to CloudWatch for containerized applications running on either Amazon ECS and AWS Fargate or Amazon EKS.

To trace the execution times or errors from calls between services, you can use AWS X-Ray²². AWS X-Ray lets you understand how your application and its underlying services are performing so you can identify and troubleshoot the root cause of performance issues and errors. X-Ray provides an end-to-end view of requests as they travel through your application, and shows a map of your application's underlying components.

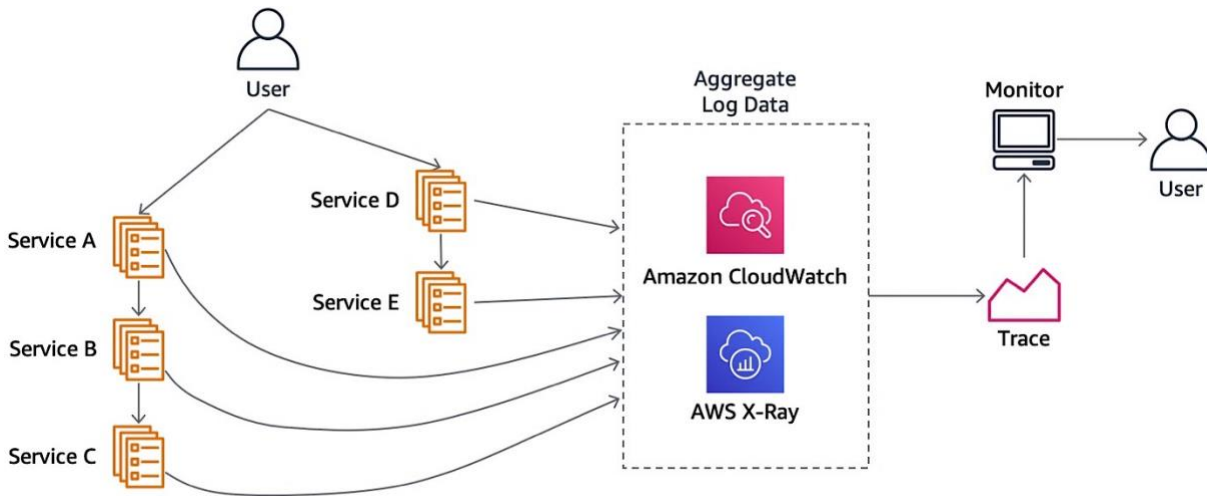


Figure 17 – Example of an architecture with log aggregation using Amazon CloudWatch and AWS X-Ray

Polyglot Persistence

In microservice architectures, each service should expose a public API and hide implementation details from other services. With this architecture, as long as the team that builds a given service maintains the API contract, it can freely modify the internals of a service without worrying whether other services depend on the modified code. Teams can also make deployments to their own services when they need to and can choose to implement their service with their preferred programming languages and databases. With *polyglot persistence*, you choose the correct data storage technology based on the data access patterns and other requirements of a given service.

If every service team has to use the same data storage technology, they can encounter implementation challenges or poor performance if that data store is not a good fit for a given situation. When teams are allowed to choose the data store that is the best fit for their requirements, they can implement their services more easily and achieve better performance and scalability.

AWS offers several data storage services that enable polyglot persistence, as summarized in the following table.

Table 1 – AWS data storage services for polyglot persistence

Data Store	Features
Amazon DynamoDB	<p>A key-value and document database that delivers single-digit millisecond performance at any scale. It's a fully managed, multi-region, multi-master, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications. DynamoDB can handle more than 10 trillion requests per day and can support peaks of more than 20 million requests per second.</p>
Amazon Aurora and Amazon Relational Database Service (RDS)	<p>Amazon Aurora is a MySQL and PostgreSQL-compatible relational database built for the cloud, that combines the performance and availability of traditional enterprise databases with the simplicity and cost-effectiveness of open source database.</p> <p>Amazon Relational Database Service (Amazon RDS)²³ makes it easy to set up, operate, and scale a relational database in the cloud. It provides cost-efficient and resizable capacity, while automating time-consuming administration tasks, such as hardware provisioning, database setup, patching and backups. It enables you to focus on your applications so you can give them the fast performance, high availability, security, and compatibility they need.</p>
Amazon ElastiCache	<p>Amazon ElastiCache offers fully managed Redis and Memcached. Seamlessly deploy, run, and scale popular open-source compatible, in-memory data stores. Build data-intensive apps or improve the performance of your existing apps by retrieving data from high throughput and low latency in-memory data stores.</p>
Amazon EBS	<p>Amazon Elastic Block Store (EBS) is an easy to use, high performance block storage service designed for use with Amazon Elastic Compute Cloud (EC2) for both throughput and transaction intensive workloads at any scale.</p> <p>Amazon EBS volume data is replicated across multiple servers in an Availability Zone to prevent the loss of data from the failure of any single component. Amazon EBS volumes offer the consistent and low-latency performance needed to run your workloads. With Amazon EBS, you can scale your usage up or down within minutes—all while paying a low price for only what you provision.</p>

Data Store	Features
Amazon EFS	Amazon Elastic File System (Amazon EFS) provides a simple, scalable, elastic file system for Linux-based workloads for use with AWS Cloud services and on-premises resources. It is built to scale on demand to petabytes without disrupting applications, growing and shrinking automatically as you add and remove files, so your applications have the storage they need when they need it. It is designed to provide massively parallel shared access to thousands of Amazon EC2 instances, which enables your applications to achieve high levels of aggregate throughput and IOPS with consistent low latencies.
Amazon S3	Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. This means customers of all sizes and industries can use it to store and protect any amount of data for a range of use cases, such as websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics.

Continuous Integration and Continuous Delivery on AWS

Because continuous integration (CI) and continuous delivery (CD) are critical to recognizing the value of modern application development, it is important carefully consider how to implement these best practices on AWS. AWS offers several services to help you deliver modern applications quickly, as discussed in the [Automating Deployment with CI/CD](#) section. Because these services are fully managed, your development teams can focus on automating deployments and rapidly delivering new functionality instead of the undifferentiated heavy lifting of maintaining and securing CI servers.

CI/CD Services on AWS

You can use the following AWS services for CI/CD deployments in the AWS Cloud.

AWS Cloud9

AWS Cloud9²⁴ is a cloud-based integrated development environment (IDE) that you can use to write, run, and debug your code with only a browser. It includes a code editor, debugger, and terminal. AWS Cloud9 includes essential tools for popular programming languages, including JavaScript, Python, and PHP, so you don't have to install files or configure your development machine to start new projects.

Because your AWS Cloud9 IDE is cloud-based, you can work on projects from your office, home, or anywhere you have an internet-connected machine. AWS Cloud9 also provides a seamless experience for developing serverless applications, which enables you to easily define resources, debug, and switch between local and remote execution of serverless applications. With AWS Cloud9, you can quickly share your development environment with your team, which enables you to pair program and track each other's inputs in real time.

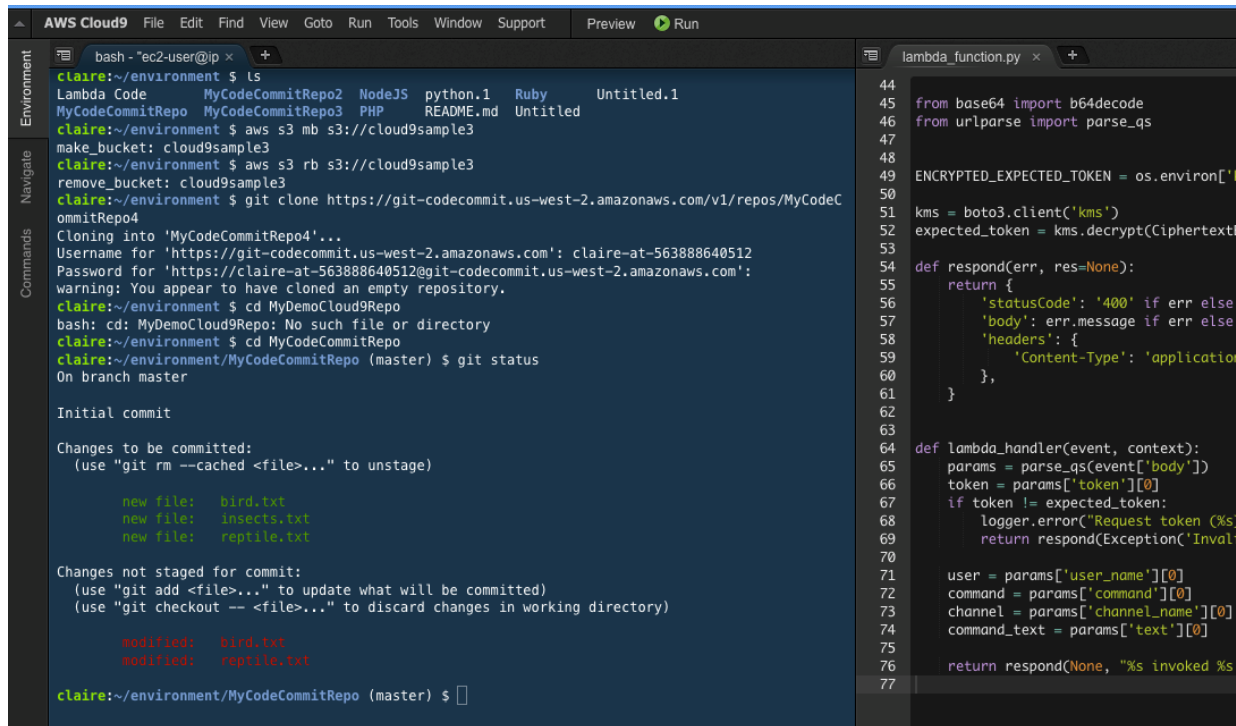


Figure 18 – Example of code in AWS Cloud9

AWS CodeStar

AWS CodeStar²⁵ enables you to quickly develop, build, and deploy applications in the AWS Cloud. AWS CodeStar provides a unified user interface, which enables you to easily manage your software development activities in one place. With AWS CodeStar, you can set up your entire continuous delivery toolchain in minutes, so you can start releasing code faster. AWS CodeStar makes it easy for your whole team to work together securely. You can easily manage access and add owners, contributors, and viewers to your projects. Each AWS CodeStar project includes a project management dashboard that you can use to easily track progress across your entire software development process, from your backlog of work items to your teams' recent code deployments.

AWS CodePipeline

AWS CodePipeline²⁶ is a fully managed continuous delivery service that helps you automate your release pipelines for fast and reliable application and infrastructure updates. CodePipeline automates the build, test, and deploy phases of your release process each time there is a code change, based on the release model you define. This enables you to rapidly and reliably deliver features and updates.

AWS CodeCommit

AWS CodeCommit²⁷ is a fully-managed source control service that hosts secure Git-based repositories.²⁸ It makes it easy for teams to collaborate on code in a secure and highly scalable ecosystem. CodeCommit eliminates the need to operate your own source control system or worry about scaling its infrastructure. You can use CodeCommit to securely store anything from source code to binaries, and it works seamlessly with your existing Git tools.

AWS CodeBuild

AWS CodeBuild²⁹ is a fully managed continuous integration service that compiles source code, runs tests, and produces software packages that are ready to deploy. With CodeBuild, you don't need to provision, manage, and scale your own build servers. CodeBuild scales continuously and processes multiple builds concurrently, so your builds are not left waiting in a queue. You can get started quickly by using prepackaged build environments, or you can create custom build environments that use your own build tools.

AWS CodeDeploy

AWS CodeDeploy³⁰ is a fully managed deployment service that automates software deployments to a variety of computing services, such as Amazon Elastic Compute Cloud (Amazon EC2), AWS Fargate, AWS Lambda, and your on-premises servers. With AWS CodeDeploy, you can rapidly release new features and avoid downtime during application deployment. AWS CodeDeploy also handles the complexity of updating your applications. You can use AWS CodeDeploy to automate software deployments, which eliminates the need for error-prone manual operations. The service scales to match your deployment needs.

AWS Amplify Console

The AWS Amplify Console³¹ provides a Git-based workflow to deploy and host full-stack serverless web applications. A full-stack serverless application consists of a backend built with cloud resources, such as GraphQL³² or REST APIs, file and data storage, and a frontend built with single page application frameworks, such as React³³, Angular³⁴, Vue³⁵, or Gatsby³⁶. Full-stack serverless web application functionality is often spread across frontend code that runs in the browser and backend business logic that runs in the cloud. This makes application deployment complex and time consuming because you must carefully coordinate release cycles to make sure that your frontend and backend are compatible, and that new features do not break your production customers. The Amplify Console accelerates your application release cycle by providing

a simple workflow to deploy full-stack serverless applications. You connect your application's code repository to Amplify Console, and changes to your frontend and backend are deployed in a single workflow on every code commit.

CI/CD Patterns for Different Application Types

You can use a CI/CD pattern for each major type of modern application that you might deploy in the AWS Cloud. You can implement CI/CD quickly using AWS-native development tools without worrying about the heavy lifting of setting up and maintaining a complicated CI environment. The following are some examples of how you can use CI/CD patterns in the AWS Cloud.

Deploy a Single-Page Application

Single-page applications (SPAs) are applications that consist of static content (HTML, CSS, JavaScript, and media) that is downloaded to the browser, from which calls are made to backend APIs. You can use the AWS Amplify Console to quickly build and release SPAs. AWS Amplify console can automatically detect when new code is pushed to repositories such as GitHub³⁷ or AWS CodeCommit, deploy the static frontend content to Amazon Simple Storage Service (Amazon S3), then deliver the content to your users through Amazon CloudFront³⁸, a content delivery network. The Amplify Console can also deploy changes to serverless backends with GraphQL and REST APIs, authentication, analytics, and storage created by the Amplify CLI.

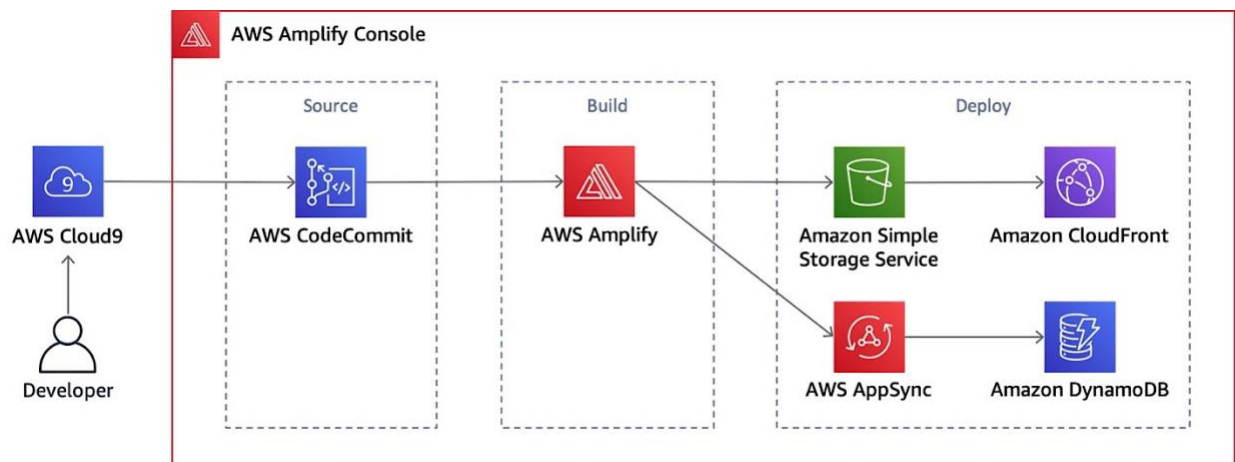


Figure 19 – Example architecture of deployment for a single-page application

Deploy to Containers

Using AWS CodePipeline, you can continuously deploy to the Amazon Elastic Container Service (Amazon ECS) container orchestration service with minimal configuration. In the source stage, AWS CodePipeline automatically detects changes in the source code repository. In the build stage, it builds Docker images using AWS CodeBuild and pushes them to a Docker repository, such as Amazon Elastic Container Registry (ECR)³⁹. Finally, AWS CodePipeline deploys to Amazon ECS.

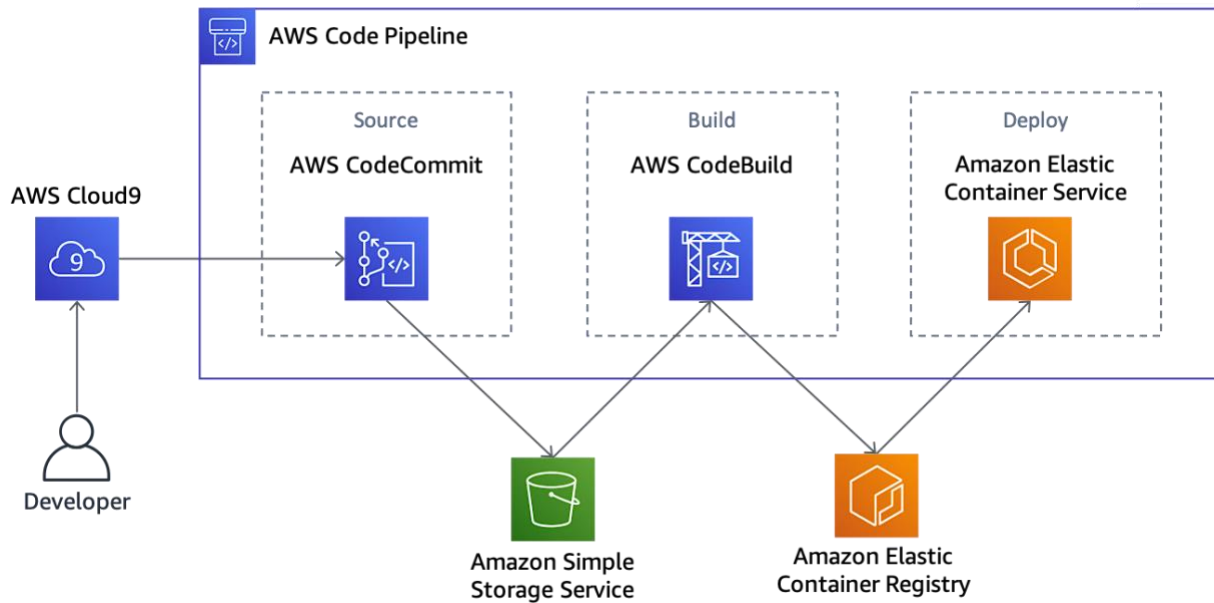


Figure 20 – Example architecture of deployment to containers

Deploy to Containers (Blue/Green Deployment)

Amazon ECS and AWS CodeDeploy also support blue/green deployment to containers. AWS CodeDeploy uses Application Load Balancers (ALBs)—a type of Amazon Elastic Load Balancing⁴⁰—to automate blue/green deployments by switching traffic smoothly between two parallel target groups.

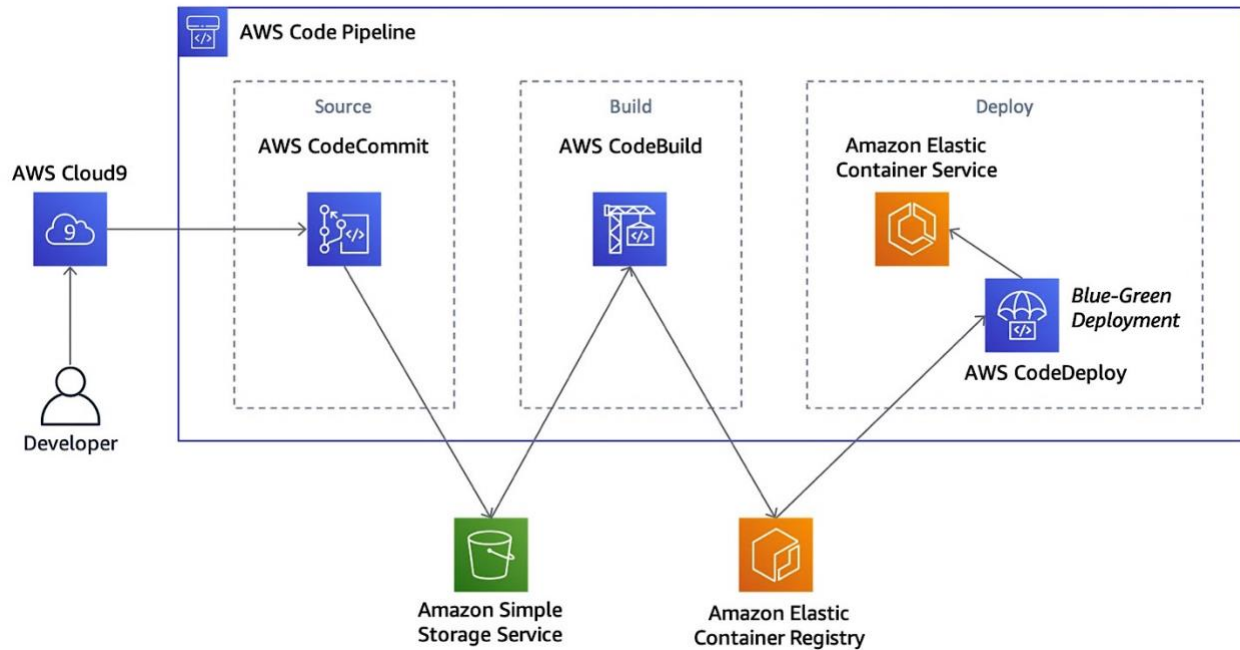


Figure 21 – Example architecture of blue/green deployment to containers

Canary Deployments to AWS Lambda

AWS CodeDeploy also supports canary deployments to AWS Lambda. AWS CodeDeploy uses Lambda's traffic shifting capabilities to automate the gradual rollout of new function versions. This enables you to gradually shift traffic between two versions, and helps you reduce the risk and limit the impact of new Lambda deployments.

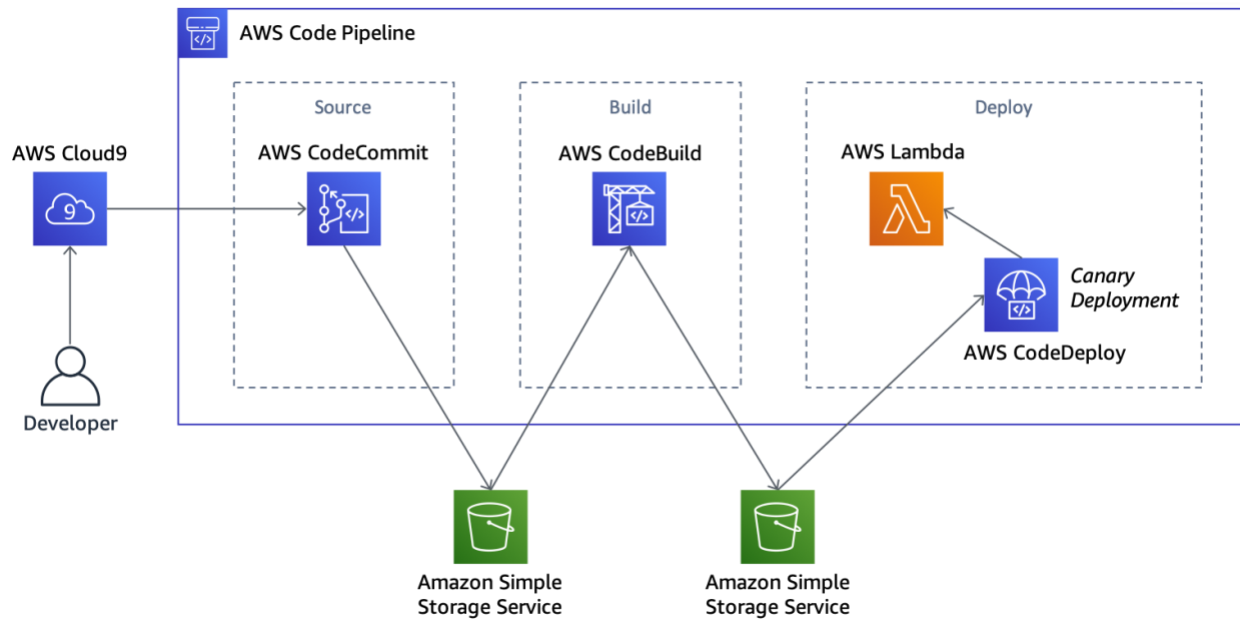


Figure 22 – Example architecture of a canary deployment in the AWS Cloud

When you perform AWS Lambda deployments with AWS CodeDeploy, you can use one of the following predefined deployment configuration options or you can create your own custom configuration. All of these options can also be used to deploy applications based on the Serverless Application Model (SAM).

Table 2 – Predefined deployment configuration options for canary deployments with AWS Lambda and AWS CodeDeploy

Deployment Configuration	Description
CodeDeployDefault.LambdaCanary10Percent5Minutes	Shifts 10 percent of traffic in the first increment. The remaining 90 percent is deployed 15 minutes later.
CodeDeployDefault.LambdaCanary10Percent10Minutes	Shifts 10 percent of traffic in the first increment. The remaining 90 percent is deployed 10 minutes later.
CodeDeployDefault.LambdaCanary10Percent15Minutes	Shifts 10 percent of traffic in the first increment. The remaining 90 percent is deployed 15 minutes later.
CodeDeployDefault.LambdaCanary10Percent30Minutes	Shifts 10 percent of traffic in the first increment. The remaining 90 percent is deployed 30 minutes later.
CodeDeployDefault.LambdaLinear10PercentEvery1Minute	Shifts 10 percent of traffic every minute until all traffic is shifted.
CodeDeployDefault.LambdaLinear10PercentEvery2Minutes	Shifts 10 percent of traffic every two minutes until all traffic is shifted.
CodeDeployDefault.LambdaLinear10PercentEvery3Minutes	Shifts 10 percent of traffic every three minutes until all traffic is shifted.
CodeDeployDefault.LambdaLinear10PercentEvery10Minutes	Shifts 10 percent of traffic every 10 minutes until all traffic is shifted.
CodeDeployDefault.LambdaAllAtOnce	Shifts all traffic to the updated Lambda functions at once.

Conclusion

Modern companies must broaden their reach across the globe and invest in digital initiatives to beat the competition. As user interaction with digital products evolves, the customer experience must get better and satisfy an increasingly diverse pool of users. To satisfy users' high expectations, businesses must not fear failure, but must constantly experiment and incorporate user feedback into their products.

Modern application development is a mindset and a methodology to enable rapid updates and releases. Development teams that embrace these modern practices eliminate undifferentiated heavy lifting by automating repetitive tasks, using managed services wherever possible, and spending most of their time building products that delight their customers.

Successfully adopting modern application development best practices enables your organization to experiment and innovate rapidly, and using native AWS services to implement these practices lets you move even faster.

Contributors

Contributors to this document include:

- Atsushi Fukui, Solutions Architect, Amazon Web Services
- Kevin Bell, Solutions Architect, Amazon Web Services

Further Reading

For more information, see the following resources.

AWS Services

- Amazon API Gateway
<https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html>
- AWS Cloud Map
<https://docs.aws.amazon.com/cloud-map/latest/dg/what-is-cloud-map.html>
- AWS Lambda
<https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
- Amazon Kinesis Data Streams
<https://docs.aws.amazon.com/streams/latest/dev/introduction.html>
- Amazon Simple Queue Service
<https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html>
- Amazon Simple Notification Service
<https://docs.aws.amazon.com/sns/latest/dg/welcome.html>
- Amazon Elastic Container Service
<https://docs.aws.amazon.com/AmazonECS/latest/userguide/Welcome.html>
- Amazon EKS
<https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html>
- AWS CodePipeline
<https://docs.aws.amazon.com/codepipeline/latest/userguide/welcome.html>
- AWS CodeCommit
<https://docs.aws.amazon.com/codecommit/latest/userguide/welcome.html>
- AWS CodeBuild
<https://docs.aws.amazon.com/codebuild/latest/userguide/welcome.html>
- AWS CodeDeploy
<https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html>

- Blue/Green Deployments from AWS CodeDeploy to Amazon ECS
<https://docs.aws.amazon.com/AmazonECS/latest/userguide/deployment-type-bluegreen.html>
- AWS CodeStar
<https://docs.aws.amazon.com/codestar/latest/userguide/welcome.html>
- AWS Cloud9
<https://docs.aws.amazon.com/cloud9/latest/user-guide/welcome.html>
- Messaging Services
<https://aws.amazon.com/messaging/>
- Serverless Services
<https://aws.amazon.com/serverless/>

Whitepapers

- Microservices on AWS
<https://d1.awsstatic.com/whitepapers/microservices-on-aws.pdf>
- Introduction to DevOps on AWS
https://d1.awsstatic.com/whitepapers/AWS_DevOps.pdf
- Infrastructure as Code
<https://d1.awsstatic.com/whitepapers/infrastructure-as-code.pdf>
- Practicing Continuous Integration and Continuous Delivery on AWS
<https://d1.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf>

Video

Choosing the Right Messaging Service for Your Distributed App (API305)

<https://www.youtube.com/watch?v=4-JmX6MIDDI>

Document Revisions

Date	Description
October 2019	First publication

Notes

- 1 Amazon EC2 – <https://aws.amazon.com/ec2/>
- 2 Microservices – <https://martinfowler.com/articles/microservices.html>
- 3 AWS Lambda – <https://aws.amazon.com/lambda/>
- 4 AWS Fargate – <https://aws.amazon.com/fargate/>
- 5 Amazon S3 – <https://aws.amazon.com/s3/>
- 6 Amazon DynamoDB – <https://aws.amazon.com/dynamodb/>
- 7 Amazon Aurora Serverless – <https://aws.amazon.com/rds/aurora/serverless/>
- 8 Serverless – <https://aws.amazon.com/serverless/>
- 9 AWS CloudFormation – <https://aws.amazon.com/cloudformation/>
- 10 AWS Serverless Application Model – <https://aws.amazon.com/serverless/sam/>
- 11 AWS CDK – <https://docs.aws.amazon.com/cdk/latest/guide/what-is.html>
- 12 Amazon API Gateway – <https://aws.amazon.com/api-gateway/>
- 13 Envoy Proxy – <https://www.envoyproxy.io/>
- 14 Amazon Kinesis – <https://aws.amazon.com/kinesis/>
- 15 Amazon Simple Queue Service – <https://aws.amazon.com/sqs/>
- 16 Amazon MQ – <https://aws.amazon.com/amazon-mq/>
- 17 Amazon MSK – <https://aws.amazon.com/msk/>
- 18 Amazon CloudWatch – <https://aws.amazon.com/cloudwatch/>
- 19 Fluentd – <https://www.fluentd.org/>
- 20 Fluent Bit – <https://fluentbit.io/>
- 21 Using Container Insights – <https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/ContainerInsights.html>
- 22 AWS X-Ray – <https://aws.amazon.com/xray/>
- 23 Amazon RDS – <https://aws.amazon.com/rds/>
- 24 Amazon RDS – <https://aws.amazon.com/rds/>
- 25 AWS CodeStar – <https://aws.amazon.com/codestar/>

- 26 AWS CodePipeline – <https://aws.amazon.com/codepipeline/>
- 27 AWS CodeCommit <https://aws.amazon.com/codecommit/>
- 28 Git – <https://git-scm.com/>
- 29 AWS CodeBuild – <https://aws.amazon.com/codebuild/>
- 30 AWS CodeDeploy – <https://aws.amazon.com/codedeploy/>
- 31 AWS Amplify Console – <https://aws.amazon.com/amplify/console/>
- 32 GraphQL – <https://graphql.org/>
- 33 React – <https://reactjs.org/>
- 34 Angular – <https://angular.io/>
- 35 Vue – <https://vuejs.org/index.html>
- 36 Gatsby – <https://www.gatsbyjs.org/>
- 37 GitHub – <https://github.com/>
- 38 Amazon CloudFront – <https://aws.amazon.com/cloudfront/>
- 39 Amazon Elastic Container Registry – <https://aws.amazon.com/ecr/>
- 40 Amazon Elastic Load Balancing – <https://aws.amazon.com/elasticloadbalancing/>