Implementing Microservices on AWS

First Published December 1, 2016

Updated November 9, 2021

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.

Contents

Abstract

Microservices are an architectural and organizational approach to software development created to speed up deployment cycles, foster innovation and ownership, improve maintainability and scalability of software applications, and scale organizations delivering software and services by using an agile approach that helps teams work independently. With a microservices approach, software is composed of small services that communicate over well-defined application programming interfaces (APIs) that can be deployed independently. These services are owned by small autonomous teams. This agile approach is key to successfully scale your organization.

Three common patterns have been observed when AWS customers build microservices: API driven, event driven, and data streaming. This whitepaper introduces all three approaches and summarizes the common characteristics of microservices, discusses the main challenges of building microservices, and describes how product teams can use Amazon Web Services (AWS) to overcome these challenges.

Due to the rather involved nature of various topics discussed in this whitepaper, including data store, asynchronous communication, and service discovery, the reader is encouraged to consider specific requirements and use cases of their applications, in addition to the provided guidance, prior to making architectural choices.

Introduction

Microservices architectures are not a completely new approach to software engineering, but rather a combination of various successful and proven concepts such as:

- Agile software development
- Service-oriented architectures
- API-first design
- Continuous integration/continuous delivery (CI/CD)

In many cases, design patterns of the Twelve-Factor App are used for microservices.

This whitepaper first describes different aspects of a highly scalable, fault-tolerant microservices architecture (user interface, microservices implementation, and data store) and how to build it on AWS using container technologies. It then recommends the AWS services for implementing a typical serverless microservices architecture to reduce operational complexity.

Serverless is defined as an operational model by the following tenets:

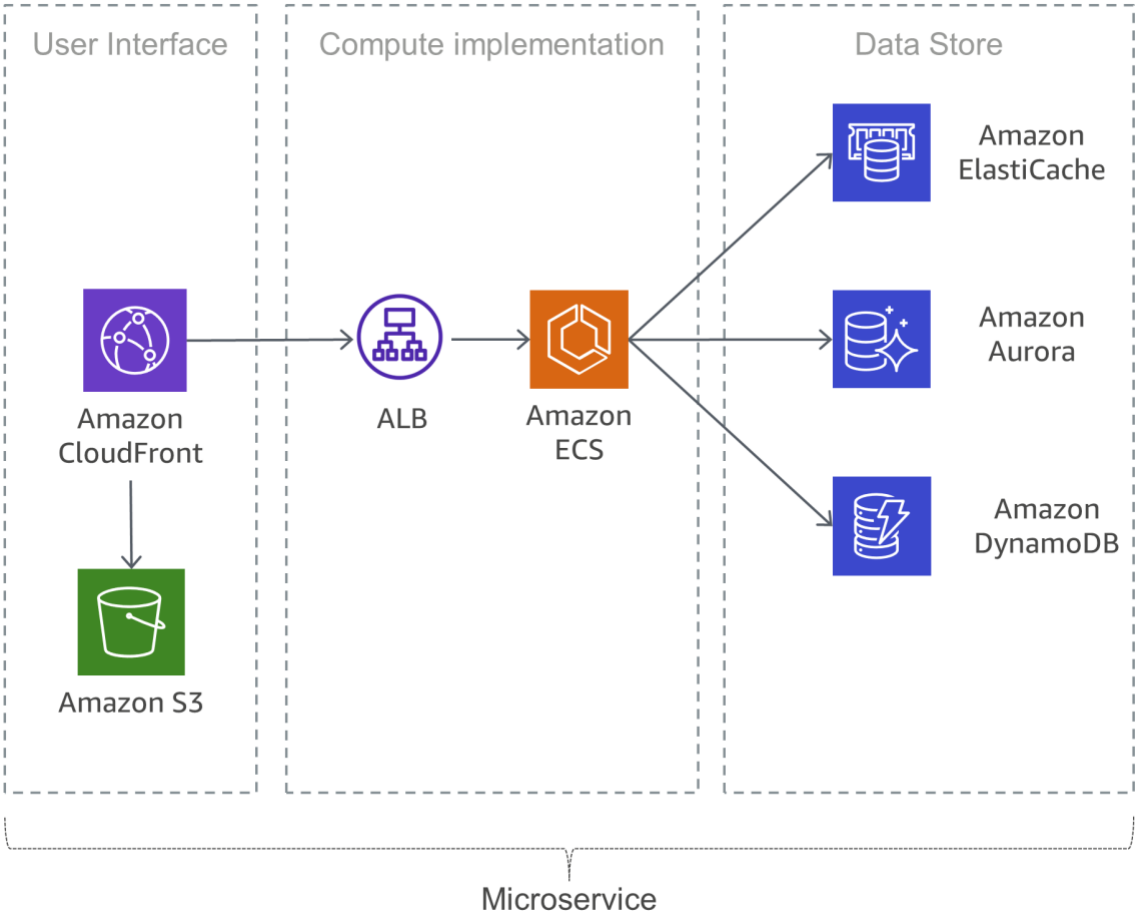
- No infrastructure to provision or manage
- Automatically scaling by unit of consumption
- *Pay for value* billing model
- Built-in availability and fault tolerance

Finally, this whitepaper covers the overall system and discusses the cross-service aspects of a microservices architecture, such as distributed monitoring and auditing, data consistency, and asynchronous communication.

This whitepaper only focuses on workloads running in the AWS Cloud. It doesn't cover hybrid scenarios or migration strategies. For more information about migration, refer to the Container Migration Methodology whitepaper.

Microservices architecture on AWS

Typical monolithic applications are built using different layers—a user interface (UI) layer, a business layer, and a persistence layer. A central idea of a microservices architecture is to split functionalities into cohesive *verticals*—not by technological layers, but by implementing a specific domain. The following figure depicts a reference architecture for a typical microservices application on AWS.



Typical microservices application on AWS

User interface

Modern web applications often use JavaScript frameworks to implement a single-page application that communicates with a representational state transfer (REST) or RESTful

API. Static web content can be served using Amazon Simple Storage Service (Amazon S3) and Amazon CloudFront.

Because clients of a microservice are served from the closest edge location and get responses either from a cache or a proxy server with optimized connections to the origin, latencies can be significantly reduced. However, microservices running close to each other don't benefit from a content delivery network. In some cases, this approach might actually add additional latency. A best practice is to implement other caching mechanisms to reduce chattiness and minimize latencies. For more information, refer to the Chattiness topic.

Microservices

APIs are the front door of microservices, which means that APIs serve as the entry point for applications logic behind a set of programmatic interfaces, typically a RESTful web services API. This API accepts and processes calls from clients, and might implement functionality such as traffic management, request filtering, routing, caching, authentication, and authorization.

Microservices implementation

AWS has integrated building blocks that support the development of microservices. Two popular approaches are using AWS Lambda and Docker containers with AWS Fargate.

With AWS Lambda, you upload your code and let Lambda take care of everything required to run and scale the implementation to meet your actual demand curve with high availability. No administration of infrastructure is needed. Lambda supports several programming languages and can be invoked from other AWS services or be called directly from any web or mobile application. One of the biggest advantages of AWS Lambda is that you can move quickly: you can focus on your business logic because security and scaling are managed by AWS. Lambda's opinionated approach drives the scalable platform.

A common approach to reduce operational efforts for deployment is container-based deployment. Container technologies, like Docker, have increased in popularity in the last few years due to benefits like portability, productivity, and efficiency. The learning curve with containers can be steep and you have to think about security fixes for your Docker images and monitoring. Amazon Elastic Container Service (Amazon ECS) and Amazon

Elastic Kubernetes Service (Amazon EKS) eliminate the need to install, operate, and scale your own cluster management infrastructure. With API calls, you can launch and stop Docker-enabled applications, query the complete state of your cluster, and access many familiar features like security groups, Load Balancing, Amazon Elastic Block Store (Amazon EBS) volumes, and AWS Identity and Access Management (IAM) roles.

AWS Fargate is a serverless compute engine for containers that works with both Amazon ECS and Amazon EKS. With Fargate, you no longer have to worry about provisioning enough compute resources for your container applications. Fargate can launch tens of thousands of containers and easily scale to run your most mission-critical applications.

Amazon ECS supports container placement strategies and constraints to customize how Amazon ECS places and ends tasks. A task placement constraint is a rule that is considered during task placement. You can associate attributes, which are essentially key-value pairs, to your container instances and then use a constraint to place tasks based on these attributes. For example, you can use constraints to place certain microservices based on instance type or instance capability, such as GPU-powered instances.

Amazon EKS runs up-to-date versions of the open-source Kubernetes software, so you can use all the existing plugins and tooling from the Kubernetes community. Applications running on Amazon EKS are fully compatible with applications running on any standard Kubernetes environment, whether running in on-premises data centers or public clouds. Amazon EKS integrates IAM with Kubernetes, enabling you to register IAM entities with the native authentication system in Kubernetes. There is no need to manually set up credentials for authenticating with the Kubernetes control plane. The IAM integration enables you to use IAM to directly authenticate with the control plane itself and provide fine granular access to the public endpoint of your Kubernetes control plane.

Docker images used in Amazon ECS and Amazon EKS can be stored in Amazon Elastic Container Registry (Amazon ECR). Amazon ECR eliminates the need to operate and scale the infrastructure required to power your container registry.

Continuous integration and continuous delivery (CI/CD) are best practices and a vital part of a DevOps initiative that enables rapid software changes while maintaining system stability and security. However, this is out of scope for this whitepaper. For more

information, refer to the Practicing Continuous Integration and Continuous Delivery on AWS whitepaper.

Private links

AWS PrivateLink is a highly available, scalable technology that enables you to privately connect your virtual private cloud (VPC) to supported AWS services, services hosted by other AWS accounts (VPC endpoint services), and supported AWS Marketplace partner services. You do not require an internet gateway, network address translation device, public IP address, AWS Direct Connect connection, or VPN connection to communicate with the service. Traffic between your VPC and the service does not leave the Amazon network.

Private links are a great way to increase the isolation and security of microservices architecture. A microservice, for example, could be deployed in a totally separate VPC, fronted by a load balancer, and exposed to other microservices through an AWS PrivateLink endpoint. With this setup, using AWS PrivateLink, the network traffic to and from the microservice never traverses the public internet. One use case for such isolation includes regulatory compliance for services handling sensitive data such as PCI, HIPPA and EU/US Privacy Shield. Additionally, AWS PrivateLink allows connecting microservices across different accounts and Amazon VPCs, with no need for firewall rules, path definitions, or route tables; simplifying network management. Utilizing PrivateLink, software as a service (SaaS) providers, and ISVs can offer their microservices-based solutions with complete operational isolation and secure access, as well.

Data store

The data store is used to persist data needed by the microservices. Popular stores for session data are in-memory caches such as Memcached or Redis. AWS offers both technologies as part of the managed Amazon ElastiCache service.

Putting a cache between application servers and a database is a common mechanism for reducing the read load on the database, which, in turn, may enable resources to be used to support more writes. Caches can also improve latency.

Relational databases are still very popular to store structured data and business objects. AWS offers six database engines (Microsoft SQL Server, Oracle, MySQL,

MariaDB, PostgreSQL, and Amazon Aurora) as managed services through Amazon Relational Database Service (Amazon RDS).

Relational databases, however, are not designed for endless scale, which can make it difficult and time intensive to apply techniques to support a high number of queries.

NoSQL databases have been designed to favor scalability, performance, and availability over the consistency of relational databases. One important element of NoSQL databases is that they typically don't enforce a strict schema. Data is distributed over partitions that can be scaled horizontally and is retrieved using partition keys.

Because individual microservices are designed to do one thing well, they typically have a simplified data model that might be well suited to NoSQL persistence. It is important to understand that NoSQL databases have different access patterns than relational databases. For example, it is not possible to join tables. If this is necessary, the logic has to be implemented in the application. You can use Amazon DynamoDB to create a database table that can store and retrieve any amount of data and serve any level of request traffic. DynamoDB delivers single-digit millisecond performance, however, there are certain use cases that require response times in microseconds. Amazon DynamoDB Accelerator (DAX) provides caching capabilities for accessing data.

DynamoDB also offers an automatic scaling feature to dynamically adjust throughput capacity in response to actual traffic. However, there are cases where capacity planning is difficult or not possible because of large activity spikes of short duration in your application. For such situations, DynamoDBprovides an on-demand option, which offers simple pay-per-request pricing.DynamoDB on-demand is capable of serving thousands of requests per second instantly without capacity planning.

Reducing operational complexity

The architecture previously described in this whitepaper is already using managed services, but Amazon Elastic Compute Cloud (Amazon EC2) instances still need to be managed. The operational efforts needed to run, maintain, and monitor microservices can be further reduced by using a fully serverless architecture.

API implementation

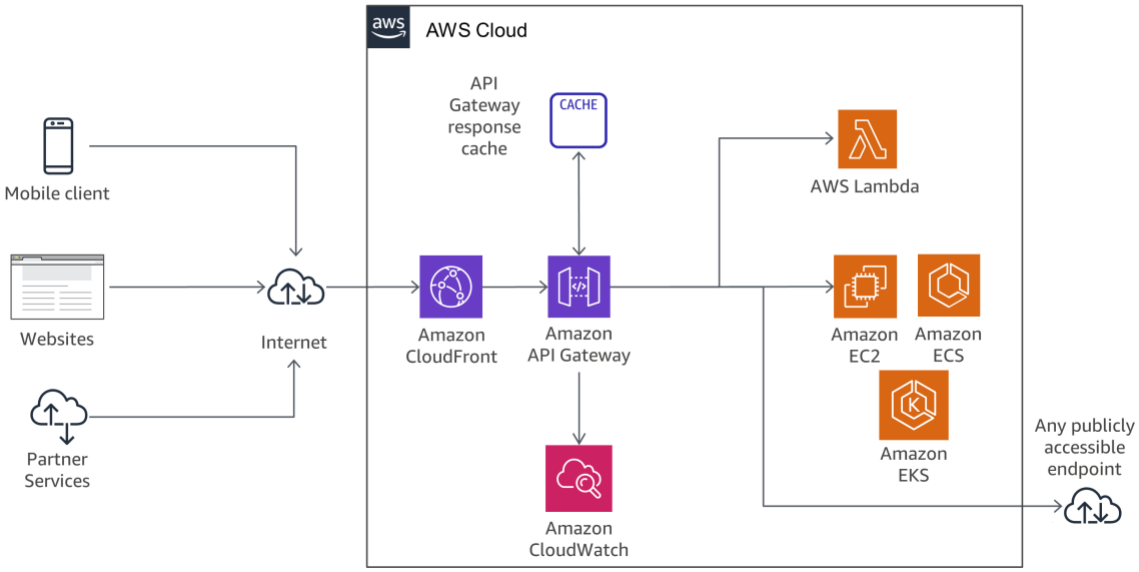
Architecting, deploying, monitoring, continuously improving, and maintaining an API can be a time-consuming task. Sometimes different versions of APIs need to be run to assure backward compatibility for all clients. The different stages of the development cycle (for example, development, testing, and production) further multiply operational efforts.

Authorization is a critical feature for all APIs, but it is usually complex to build and involves repetitive work. When an API is published and becomes successful, the next challenge is to manage, monitor, and monetize the ecosystem of third-party developers utilizing the APIs.

Other important features and challenges include throttling requests to protectthe backend services, caching API responses, handling request and response transformation, and generating API definitions and documentation with tools such as Swagger.

Amazon API Gateway addresses those challenges and reduces the operational complexity of creating and maintaining RESTful APIs. API Gateway allows you to create your APIs programmatically by importing Swagger definitions, using either the AWS API or the AWS Management Console. API Gateway serves as a front door to any web application running on Amazon EC2, Amazon ECS, AWS Lambda, or in any on-premises environment. Basically, API Gateway allows you to run APIs without having to manage servers.

The following figure illustrates how API Gateway handles API calls and interacts with other components. Requests from mobile devices, websites, or other backend services are routed to the closest CloudFront Point of Presence to minimize latency and provide optimum user experience.



API Gateway call flow

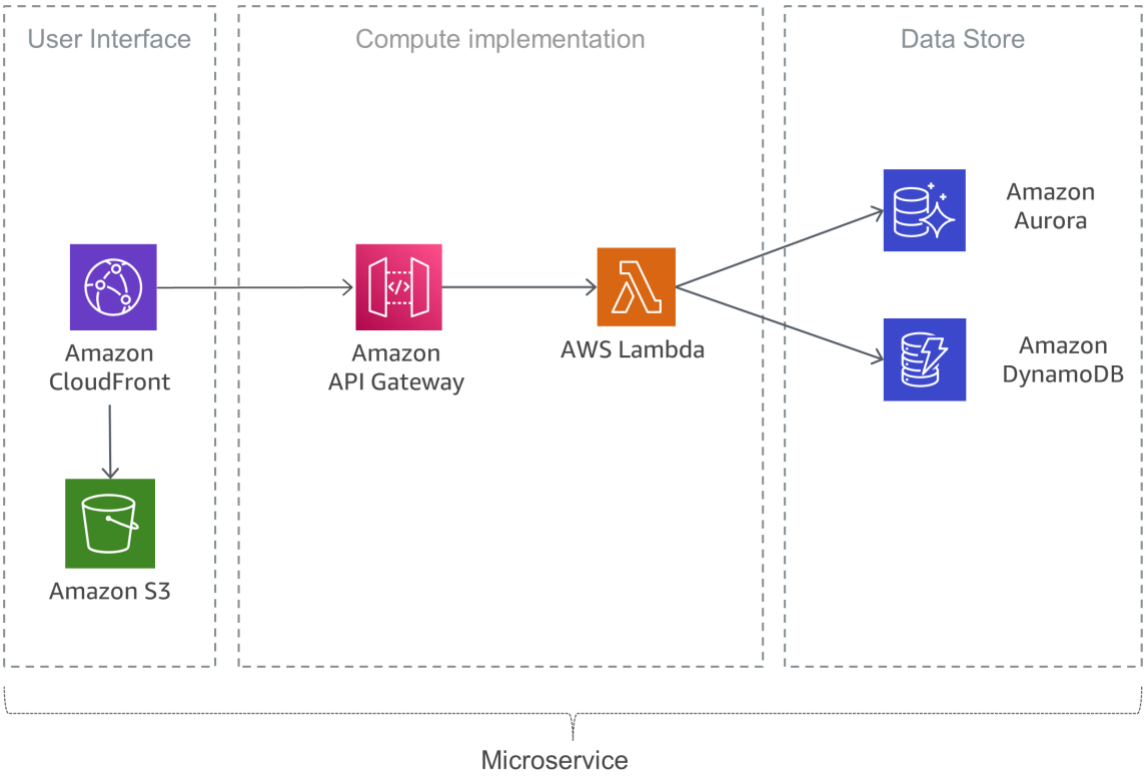
Serverless microservices

> *"No server is easier to manage than no server."* — *AWS re:Invent*

Getting rid of servers is a great way to eliminate operational complexity.

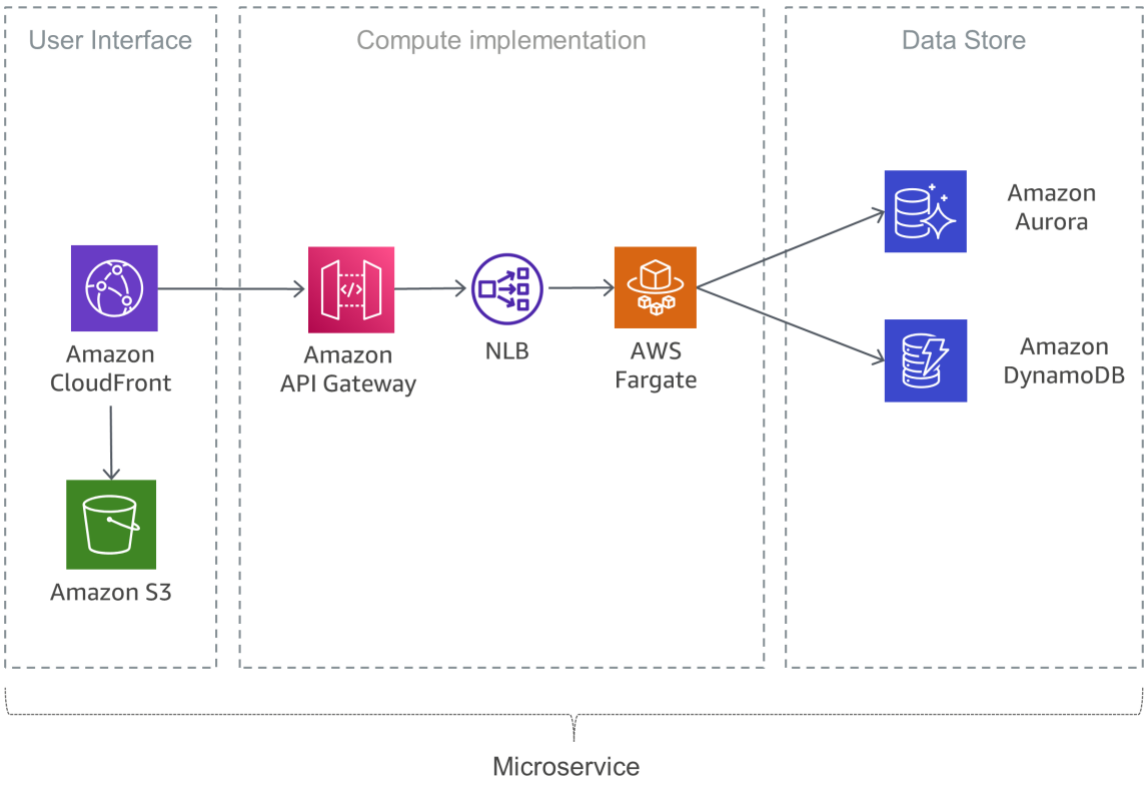
Lambda is tightly integrated with API Gateway. The ability to make synchronous calls from API Gateway to Lambda enables the creation of fully serverless applications and is described in detail in the Amazon API Gateway *Developer Guide*.

The following figure shows the architecture of a serverless microservice with AWS Lambda where the complete service is built out of managed services, which eliminates the architectural burden to design for scale and high availability, and eliminates the operational efforts of running and monitoring the microservice's underlying infrastructure.



Serverless microservice using AWS Lambda

A similar implementation that is also based on serverless services is shown in the following figure. In this architecture, Docker containers are used with Fargate, so it's not necessary to care about the underlying infrastructure. In addition to DynamoDB, Amazon Aurora Serverless is used, which is an on-demand, auto-scaling configuration for Aurora (MySQL-compatible edition), where the database will automatically start up, shut down, and scale capacity up or down based on your application's needs.



Serverless microservice using Fargate

Disaster recovery

As previously mentioned in the introduction of this whitepaper, typical microservices applications are implemented using the Twelve-Factor Application patterns. The Processes section states that “Twelve-factor processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.”

For a typical microservices architecture, this means that the main focus for disaster recovery should be on the downstream services that maintain the state of the application. For example, these can be file systems, databases, or queues, for example. When creating a disaster recovery strategy, organizations most commonly plan for the recovery time objective and recovery point objective.

Recovery time objective is the maximum acceptable delay between the interruption of service and restoration of service. This objective determines what is considered an acceptable time window when service is unavailable and is defined by the organization.

Recovery point objective is the maximum acceptable amount of time since the last data recovery point. This objective determines what is considered an acceptable loss of data between the last recovery point and the interruption of service and is defined by the organization.

For more information, refer to the Disaster Recovery of Workloads on AWS: Recovery in the Cloud whitepaper.

High availability

This section takes a closer look at high availability for different compute options.

Amazon EKS runs Kubernetes control and data plane instances across multiple Availability Zones to ensure high availability. Amazon EKS automatically detects and replaces unhealthy control plane instances, and it provides automated version upgrades and patching for them. This control plane consists of at least two API server nodes and three etcd nodes that run across three Availability Zones within a region. Amazon EKS uses the architecture of AWS Regions to maintain high availability.

Amazon ECR hosts images in a highly available and high-performance architecture, enabling you to reliably deploy images for container applications across Availability Zones. Amazon ECR works with Amazon EKS, Amazon ECS, and AWS Lambda, simplifying development to production workflow.

Amazon ECS is a regional service that simplifies running containers in a highly available manner across multiple Availability Zones within an AWS Region. Amazon ECS includes multiple scheduling strategies that place containers across your clusters based on your resource needs (for example, CPU or RAM) and availability requirements.

AWS Lambda runs your function in multiple Availability Zones to ensure that it is available to process events in case of a service interruption in a single zone. If you configure your function to connect to a virtual private cloud (VPC) in your account, specify subnets in multiple Availability Zones to ensure high availability.

Deploying Lambda-based applications

You can use AWS CloudFormation to define, deploy, and configure serverless applications.

The AWS Serverless Application Model (AWS SAM) is a convenient way to define serverless applications. AWS SAM is natively supported by CloudFormation and defines a simplified syntax for expressing serverless resources. To deploy your application, specify the resources you need as part of your application, along with their associated permissions policies in a CloudFormation template, package your deployment artifacts, and deploy the template. Based on AWS SAM, SAM Local is an AWS Command Line Interface tool that provides an environment for you to develop, test, and analyze your serverless applications locally before uploading them to the Lambda runtime.You can use SAM Local to create a local testing environment that simulates the AWS runtime environment.

Distributed systems components

After looking at how AWS can solve challenges related to individual microservices, the focus moves to on cross-service challenges, such asservice discovery, data consistency, asynchronous communication, and distributed monitoring and auditing.

Service discovery

One of the primary challenges with microservice architectures is enabling services to discover and interact with each other. The distributed characteristics of microservice architectures not only make it harder for services to communicate, but also presents other challenges, such as checking the health of those systems and announcing when new applications become available. You also must decide how and where to store meta information, such as configuration data, that can be used by applications. In this section several techniques for performing service discovery on AWS for microservices-based architectures are explored.

DNS-based service discovery

Amazon ECS now includes integrated service discovery that enables your containerized services to discover and connect with each other.

Previously, to ensure that services were able to discover and connect with each other, you had to configure and run your own service discovery system based on Amazon Route 53, AWS Lambda, and ECS event streams, or connect every service to a load balancer.

Amazon ECS creates and manages a registry of service names using the Route 53 Auto Naming API. Names are automatically mapped to a set of DNS records so that you can refer to a service by name in your code and write DNS queries to have the name resolve to the service's endpoint at runtime. You can specify health check conditions in a service's task definition and Amazon ECS ensures that only healthy service endpoints are returned by a service lookup.

In addition, you can also use unified service discovery for services managed by Kubernetes. To enable this integration, AWS contributed to the External DNS project, a Kubernetes incubator project.

Another option is to use the capabilities of AWS Cloud Map. AWS Cloud Map extends the capabilities of the Auto Naming APIs by providing a service registry for resources, such as Internet Protocols (IPs), Uniform Resource Locators (URLs), and Amazon Resource Names (ARNs), and offering an API-based service discovery mechanism with a faster change propagation and the ability to use attributes to narrow down the set of discovered resources. Existing Route 53 Auto Naming resources are upgraded automatically to AWS Cloud Map.

Third-party software

A different approach to implementing service discovery is using third-partysoftware such as HashiCorp Consul, etcd, or Netflix Eureka. All three examples are distributed, reliable key-value stores. For HashiCorp Consul, there is an AWS Quick Start that sets up a flexible, scalable AWS Cloud environment and launches HashiCorp Consul automatically into a configuration of your choice.

Service meshes

In an advanced microservices architecture, the actual application can be composed of hundreds, or even thousands, of services. Often the most complex part of the application is not the actual services themselves, but the communication between those services. Service meshes are an additional layer for handling interservice communication, which is responsible for monitoring and controlling traffic in microservices architectures. This enables tasks, like service discovery, to be completely handled by this layer.

Typically, a service mesh is split into a data plane and a control plane. The data plane consists of a set of intelligent proxies that are deployed with the application code as a

special sidecar proxy that intercepts all network communication between microservices. The control plane is responsible forcommunicating with the proxies.

Service meshes are transparent, which means that application developers don't have to be aware of this additional layer and don't have to make changes to existing application code. AWS App Mesh is a service mesh that provides application-level networking to enable your services to communicate with each other across multiple types of compute infrastructure. App Mesh standardizes how your services communicate, giving you complete visibility and ensuring high availability for your applications.

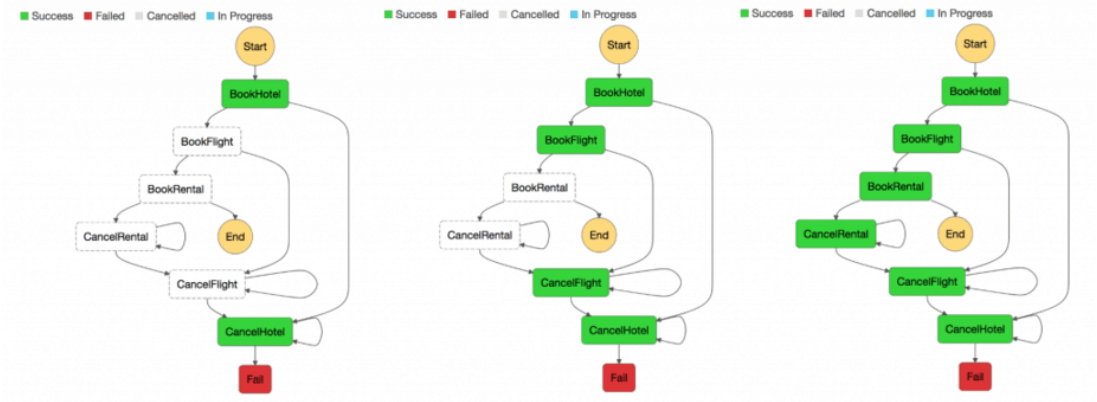
You can use App Mesh with existing or new microservices running on Amazon EC2, Fargate, Amazon ECS, Amazon EKS, and self-managed Kubernetes on AWS. App Mesh can monitor and control communications for microservices running across clusters, orchestration systems, or VPCs as a single application without any code changes.

Distributed data management

Monolithic applications are typically backed by a large relational database, which defines a single data model common to all application components. In a microservices approach, such a central database would prevent the goal of building decentralized and independent components. Each microservice component should have its own data persistence layer.

Distributed data management, however, raises new challenges. As a consequence of the CAP theorem, distributed microservice architectures inherently trade off consistency for performance and need to embrace eventual consistency.

In a distributed system, business transactions can span multiple microservices. Because they cannot use a single ACID transaction, you can end up with partial executions. In this case, we would need some control logic to redo the already processed transactions. For this purpose, the distributed Saga pattern is commonly used. In the case of a failed business transaction, Saga orchestrates a series of compensating transactions that undo the changes that were made by the preceding transactions. AWS Step Functions make it easy to implement a Saga execution coordinator as shown in the following figure.



Saga execution coordinator

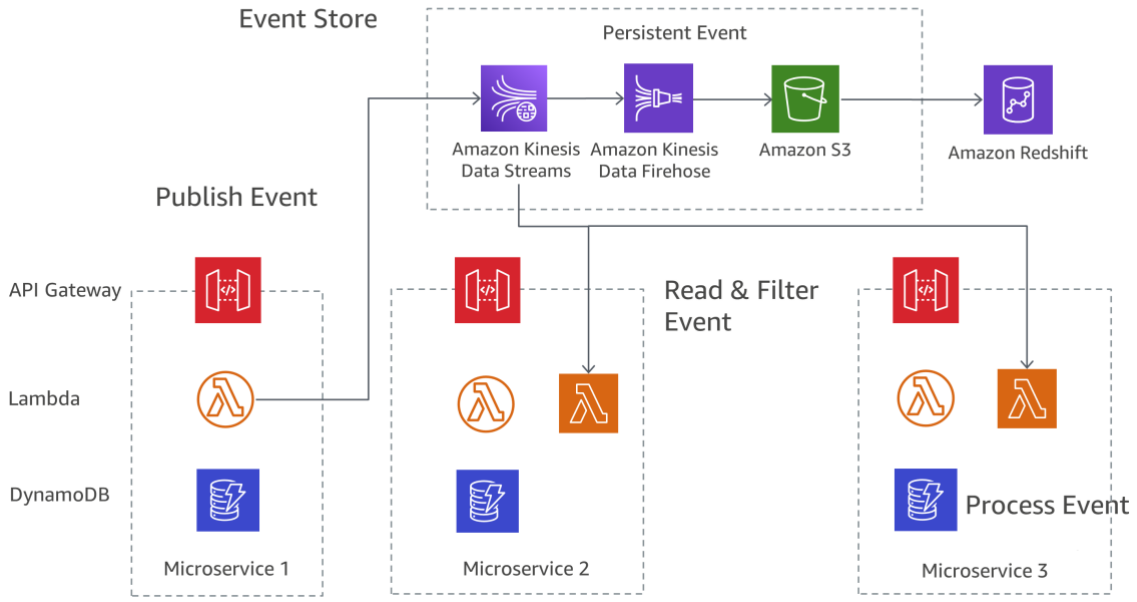
Building a centralized store of critical reference data that is curated by core data management tools and procedures provides a means for microservices to synchronize their critical data and possibly roll back state. Using AWS Lambda with scheduled Amazon CloudWatch Events you can build a simple cleanup and deduplication mechanism.

It's very common for state changes to affect more than a single microservice. In such cases, event sourcing has proven to be a useful pattern. The core idea behind event sourcing is to represent and persist every application change as an event record. Instead of persisting application state, data is stored as a stream of events. Database transaction logging and version control systems are two well-known examples for event sourcing. Event sourcing has a couple of benefits: state can be determined and reconstructed for any point in time. It naturally produces a persistent audit trail and also facilitates debugging.

In the context of microservices architectures, event sourcing enables decoupling different parts of an application by using a publish and subscribe pattern, and it feeds the same event data into different data models for separate microservices. Event sourcing is frequently used in conjunction with the Command Query Responsibility Segregation (CQRS) pattern to decouple read from write workloads and optimize both for performance, scalability, and security. In traditional data management systems, commands and queries are run against the same data repository.

The following figure shows how the event sourcing pattern can be implemented on AWS. Amazon Kinesis Data Streams serves as the main component of the central event store, which captures application changes as events and persists them on

Amazon S3. The figure depicts three different microservices, composed of API Gateway, AWS Lambda, and DynamoDB. The arrows indicate the flow of the events: when Microservice 1 experiences an event state change, it publishes an event by writing a message into Kinesis Data Streams. All microservices run their own Kinesis Data Streams application in AWS Lambda which reads a copy of the message, filters it based on relevancy for the microservice, and possibly forwards it for further processing. If your function returns an error, Lambda retries the batch until processing succeeds or the data expires. To avoid stalled shards, you can configure the event source mapping to retry with a smaller batch size, limit the number of retries, or discard records that are too old. To retain discarded events, you can configure the event source mapping to send details about failed batches to an Amazon Simple Queue Service (SQS) queue or Amazon Simple Notification Service (SNS) topic.



Event sourcing pattern on AWS

Amazon S3 durably stores all events across all microservices and is the single source of truth when it comes to debugging, recovering application state, or auditing application changes. There are two primary reasons why records may be delivered more than one time to your Kinesis Data Streams application: producer retries and consumer retries. Your application must anticipate and appropriately handle processing individual records multiple times.

Configuration management

In a typical microservices architecture with dozens of different services, each service needs access to several downstream services and infrastructure components that expose data to the service. Examples could be message queues, databases, and other microservices. One of the key challenges is to configure each service in a consistent way to provide information about the connection to downstream services and infrastructure. In addition, the configuration should also contain information about the environment in which the service is operating, and restarting the application to use new configuration data shouldn't be necessary.

The third principle of the Twelve-Factor App patterns covers this topic: "*The twelve-factor app stores config in environment variables (often shortened to env vars or env).*" For Amazon ECS, environment variables can be passed to the container by using the environment container definition parameter which maps to the `--env` option to docker run. Environment variables can be passed to your containers in bulk by using the `environmentFiles` container definition parameter to list one or more files containing the environment variables. The file must be hosted in Amazon S3. In AWS Lambda, the runtime makes environment variables available to your code and sets additional environment variables that contain information about the function and invocation request. For Amazon EKS, you can define environment variables in the env-field of the configuration manifest of the corresponding pod. A different way to use env-variables is to use a ConfigMap.

Asynchronous communication and lightweight messaging

Communication in traditional, monolithic applications is straightforward—one part of the application uses method calls or an internal event distribution mechanism to communicate with the other parts. If the same application is implemented using decoupled microservices, the communication between different parts of the application must be implemented using network communication.

REST-based communication

The HTTP/S protocol is the most popular way to implement synchronous communication between microservices. In most cases, RESTful APIs use HTTP as a

transport layer. The REST architectural style relies on stateless communication, uniform interfaces, and standard methods.

With API Gateway, you can create an API that acts as a “front door” for applications to access data, business logic, or functionality from your backend services. API developers can create APIs that access AWS or other web services, as well as data stored in the AWS Cloud. An API object defined with the API Gateway service is a group of resources and methods.

A resource is a typed object within the domain of an API and may have associated a data model or relationships to other resources. Each resource can be configured to respond to one or more methods, that is, standard HTTP verbs such as GET, POST, or PUT. REST APIs can be deployed to different stages, and versioned as well as cloned to new versions.

API Gateway 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.

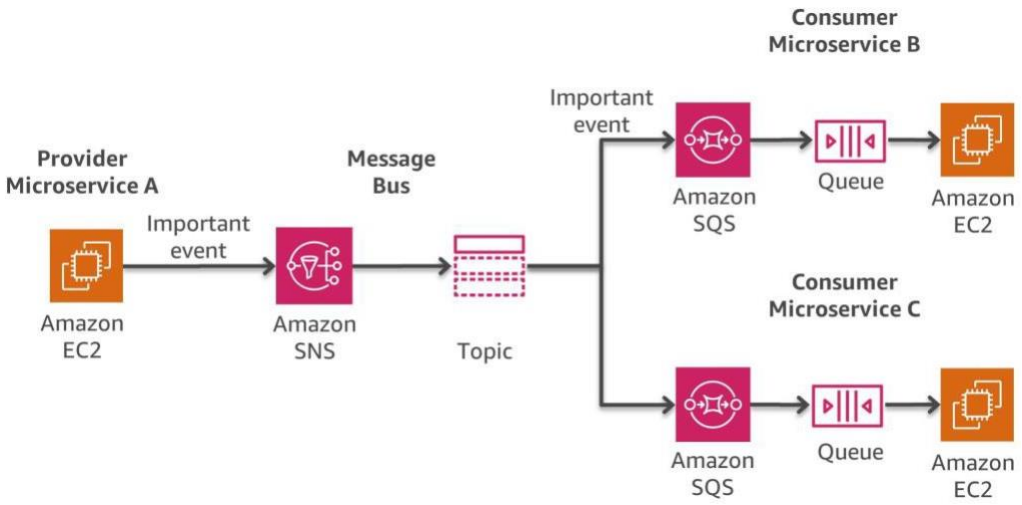
Asynchronous messaging and event passing

Message passing is an additional pattern used to implement communication between microservices. Services communicate by exchanging messages by a queue. One major benefit of this communication style is that it’s not necessary to have a service discovery and services are loosely coupled.

Synchronous systems are tightly coupled, which means a problem in a synchronous downstream dependency has immediate impact on the upstream callers. Retries from upstream callers can quickly fan-out and amplify problems.

Depending on specific requirements, like protocols, AWS offers different services which help to implement this pattern. One possible implementation uses a combination of Amazon Simple Queue Service (Amazon SQS) and Amazon Simple Notification Service (Amazon SNS).

Both services work closely together. Amazon SNS enables applications to send messages to multiple subscribers through a push mechanism. By using Amazon SNS and Amazon SQS together, one message can be delivered to multiple consumers. The following figure demonstrates the integration of Amazon SNS and Amazon SQS.



Message bus pattern on AWS

When you subscribe an SQS queue to an SNS topic, you can publish a message to the topic, and Amazon SNS sends a message to the subscribed SQS queue. The message contains subject and message published to the topic along with metadata information in JSON format.

Another option for building event-driven architectures with event sources spanning internal applications, third-party SaaS applications, and AWS services, at scale, is Amazon EventBridge. A fully managed event bus service, EventBridge receives events from disparate sources, identifies a target based on a routing rule, and delivers near real-time data to that target, including AWS Lambda, Amazon SNS, and Amazon Kinesis Streams, among others. An inbound event can also be customized, by input transformer, prior to delivery.

To develop event-driven applications significantly faster, EventBridge schema registries collect and organize schemas, including schemas for all events generated by AWS services. Customers can also define custom schemas or use an infer schema option to discover schemas automatically. In balance, however, a potential trade-off for all these features is a relatively higher latency value for EventBridge delivery. Also, the default throughput and quotas for EventBridge may require an increase, through a support request, based on use case.

A different implementation strategy is based on Amazon MQ, which can be used if existing software is using open standard APIs and protocols for messaging, including JMS, NMS, AMQP, STOMP, MQTT, and WebSocket. Amazon SQS exposes a custom

API which means, if you have an existing application that you want to migrate from—for example, an on-premises environment to AWS—code changes are necessary. With Amazon MQ this is not necessary in many cases.

Amazon MQ manages the administration and maintenance of ActiveMQ, a popular open-source message broker. The underlying infrastructure is automatically provisioned for high availability and message durability to support the reliability of your applications.

Orchestration and state management

The distributed character of microservices makes it challenging to orchestrate workflows when multiple microservices are involved. Developers might be tempted to add orchestration code into their services directly. This should be avoided because it introduces tighter coupling and makes it harder to quickly replace individual services.

You can use AWS Step Functions to build applications from individual components that each perform a discrete function. Step Functions provides a state machine that hides the complexities of service orchestration, such as error handling, serialization, and parallelization. This lets you scale and change applications quickly while avoiding additional coordination code inside services.

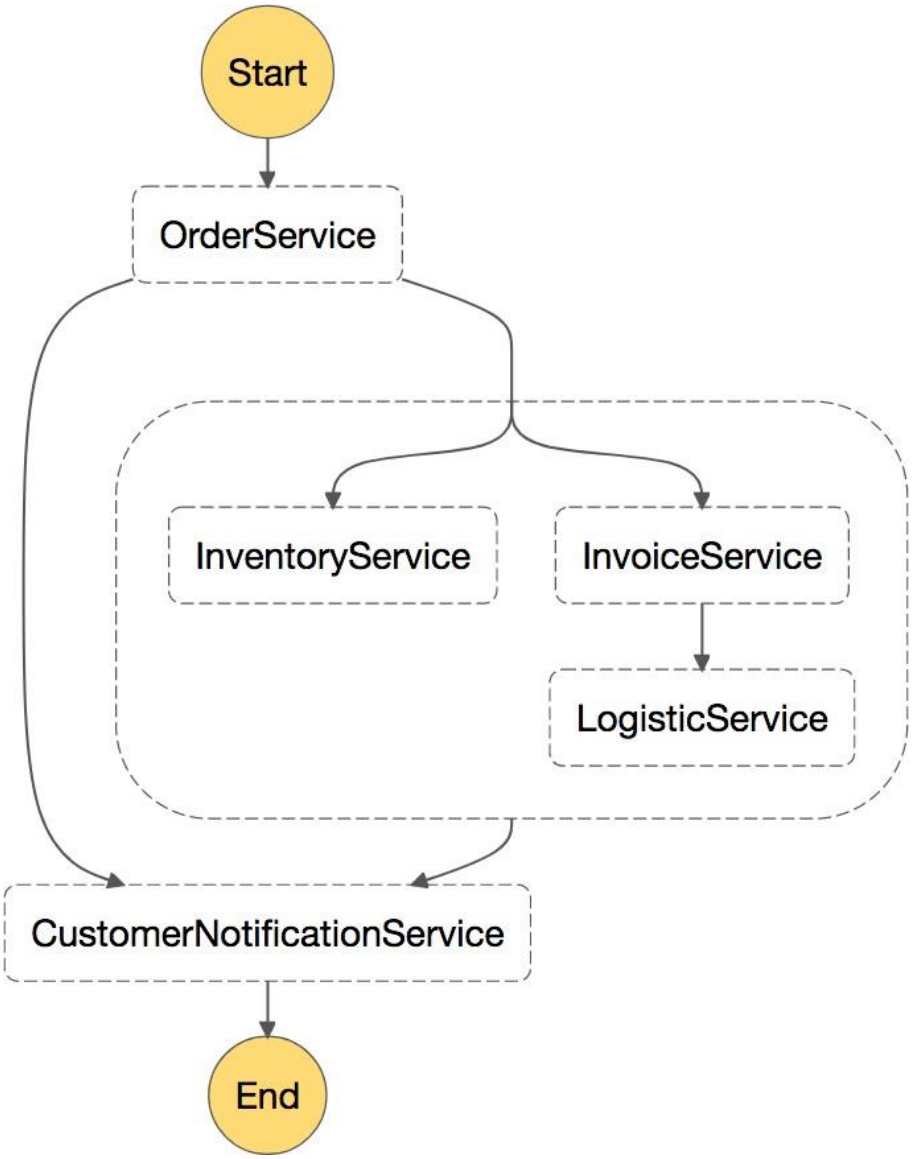
Step Functions is a reliable way to coordinate components and step through the functions of your application. Step Functions provides a graphical console to arrange and visualize the components of your application as a series of steps. This makes it easier to build and run distributed services.

Step Functions automatically starts and tracks each step and retries when there are errors, so your application executes in order and as expected. Step Functions logs the state of each step so when something goes wrong, you can diagnose and debug problems quickly. You can change and add steps without even writing code to evolve your application and innovate faster.

Step Functions is part of the AWS serverless platform and supports orchestration of Lambda functions as well as applications based on compute resources, such as Amazon EC2, Amazon EKS, and Amazon ECS, and additional services like Amazon SageMaker and AWS Glue. Step Functions manages the operations and underlying infrastructure for you to help ensure that your application is available at any scale.

To build workflows, Step Functions uses the Amazon States Language. Workflows can contain sequential or parallel steps as well as branching steps.

The following figure shows an example workflow for a microservices architecture combining sequential and parallel steps. Invoking such a workflow can be done either through the Step Functions API or with API Gateway.



An example of a microservices workflow invoked by Step Functions

Distributed monitoring

A microservices architecture consists of many different distributed parts that have to be monitored. You can use Amazon CloudWatch to collect and track metrics, centralize and monitor log files, set alarms, and automatically react to changes in your AWS environment. CloudWatch can monitor AWS resources such as Amazon EC2 instances, DynamoDB tables, and Amazon RDS DB instances, as well as custom metrics generated by your applications and services, and any log files your applications generate.

Monitoring

You can use CloudWatch to gain system-wide visibility into resource utilization, application performance, and operational health. CloudWatch provides a reliable, scalable, and flexible monitoring solution that you can start using within minutes. You no longer need to set up, manage, and scale your own monitoring systems and infrastructure. In a microservices architecture, the capability of monitoring custom metrics using CloudWatch is an additional benefit because developers can decide which metrics should be collected for each service. In addition, dynamic scaling can be implemented based on custom metrics.

In addition to Amazon Cloudwatch, you can also use CloudWatch Container Insights to collect, aggregate, and summarize metrics and logs from your containerized applications and microservices. CloudWatch Container Insights automatically collects metrics for many resources, such as CPU, memory, disk, and network and aggregate as CloudWatch metrics at the cluster, node, pod, task, and service level. Using CloudWatch Container Insights, you can gain access to CloudWatch Container Insights dashboard metrics. It also provides diagnostic information, such as container restart failures, to help you isolate issues and resolve them quickly. You can also set CloudWatch alarms on metrics that Container Insights collects.

Container Insights is available for Amazon ECS, Amazon EKS, and Kubernetes platforms on Amazon EC2. Amazon ECS support includes support for Fargate.

Another popular option, especially for Amazon EKS, is to use Prometheus. Prometheus is an open-source monitoring and alerting toolkit that is often used in combination with Grafana to visualize the collected metrics. Many Kubernetes components store metrics at `/metrics` and Prometheus can scrape these metrics at a regular interval.

Amazon Managed Service for Prometheus (AMP) is a Prometheus-compatible monitoring service that enables you to monitor containerized applications at scale. With AMP, you can use the open-source Prometheus query language (PromQL) to monitor the performance of containerized workloads without having to manage the underlying infrastructure required to manage the ingestion, storage, and querying of operational metrics. You can collect Prometheus metrics from Amazon EKS and Amazon ECS environments, using AWS Distro for OpenTelemetry or Prometheus servers as collection agents.

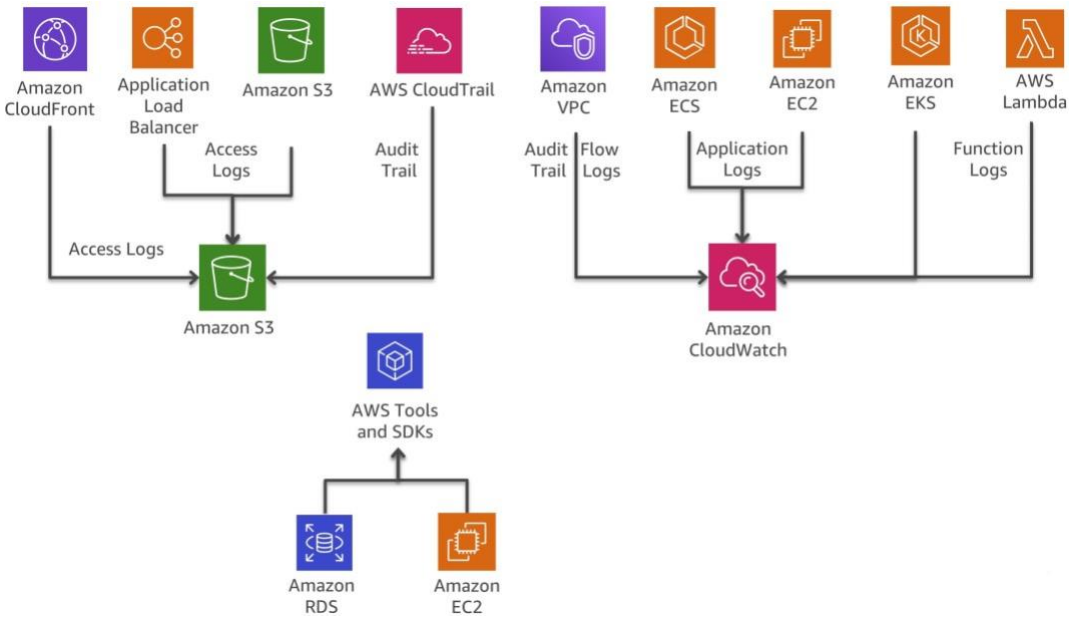
AMP is often used in combination with Amazon Managed Service for Grafana (AMG). AMG makes it easy to query, visualize, alert on and understand your metrics no matter where they are stored. With AMG, you can analyze your metrics, logs, and traces without having to provision servers, configure and update software, or do the heavy lifting involved in securing and scaling Grafana in production.

Centralizing logs

Consistent logging is critical for troubleshooting and identifying issues. Microservices enable teams to ship many more releases than ever before and encourage engineering teams to run experiments on new features in production. Understanding customer impact is crucial to gradually improving an application.

By default, most AWS services centralize their log files. The primary destinations for log files on AWS are Amazon S3 and Amazon CloudWatch Logs. For applications running on Amazon EC2 instances, a daemon is available to send log files to CloudWatch Logs. Lambda functions natively send their logoutput to CloudWatch Logs and Amazon ECS includes support for the awslogs log driver that enables the centralization of container logs to CloudWatch Logs. For Amazon EKS, either Fluent Bit or Fluentd can forward logs from the individual instances in the cluster to a centralized logging CloudWatch Logs where they are combined for higher-level reporting using Amazon OpenSearch Service and Kibana. Because of its smaller footprint and performance advantages, Fluent Bit is recommended instead of Fluentd.

The following figure illustrates the logging capabilities of some of the services. Teams are then able to search and analyze these logs using tools like Amazon OpenSearch Service and Kibana. Amazon Athena can be used to run a one-time query against centralized log files in Amazon S3.



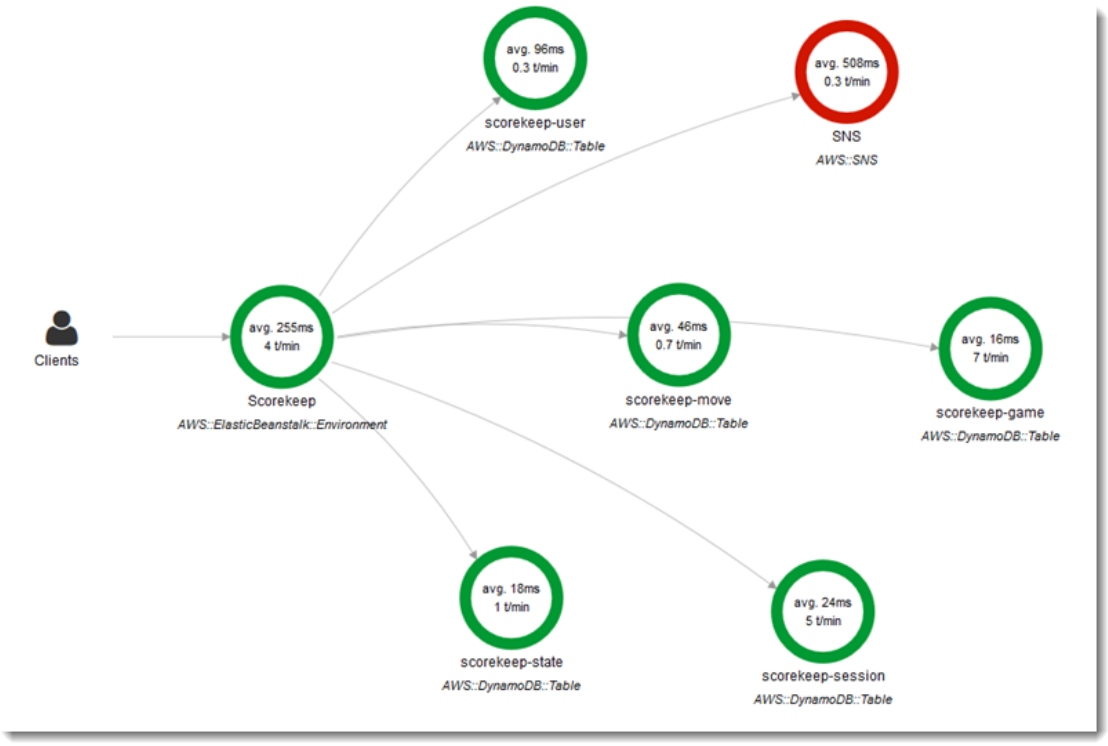
Logging capabilities of AWS services

Distributed tracing

In many cases, a set of microservices works together to handle a request. Imagine a complex system consisting of tens of microservices in which an error occurs in one of the services in the call chain. Even if every microservice is logging properly and logs are consolidated in a central system, it can be difficult to find all relevant log messages.

The central idea of AWS X-Ray is the use of correlation IDs, which are unique identifiers attached to all requests and messages related to a specific event chain. The trace ID is added to HTTP requests in specific tracing headers named `X-Amzn-Trace-Id` when the request hits the first X-Ray integrated service (for example, Application Load Balancer or API Gateway) and included in the response. Through the X-Ray SDK, any microservice can read, but can also add or update this header.

X-Ray works with Amazon EC2, Amazon ECS, AWS Lambda, and AWS Elastic Beanstalk. You can use X-Ray with applications written in Java, Node.js, and .NET that are deployed on these services.



X-Ray service map

Epsagon is fully managed SaaS that includes tracing for all AWS services, third-party APIs (through HTTP calls), and other common services such as Redis, Kafka, and Elastic. The Epsagon service includes monitoring capabilities, alerting to the most common services, and payload visibility into each and every call your code is making.

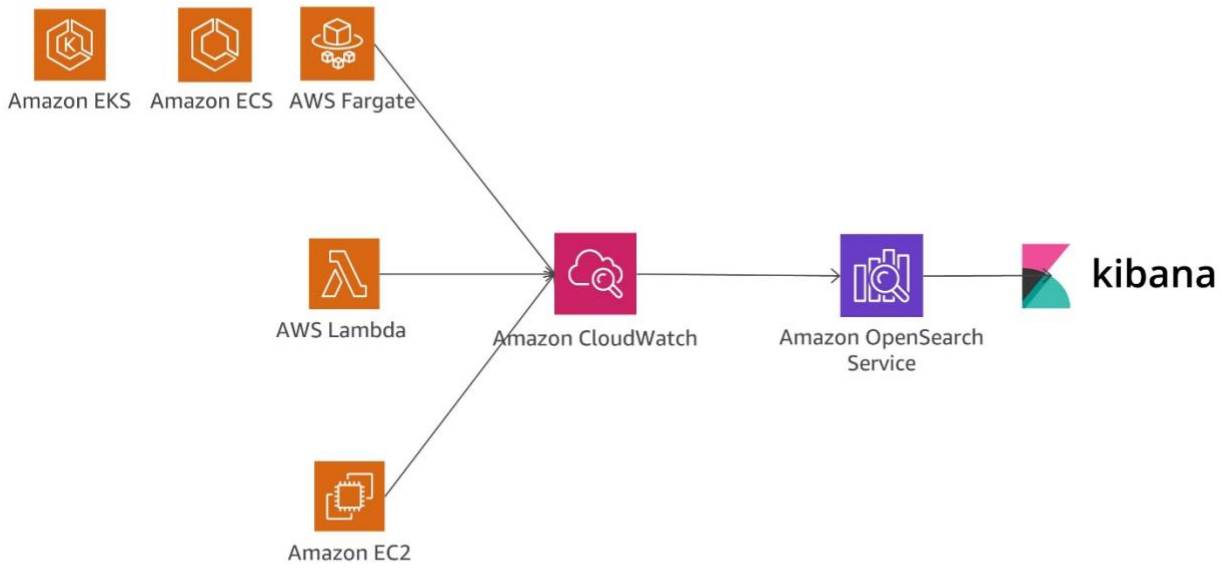
AWS Distro for OpenTelemetry is a secure, production-ready, AWS-supported distribution of the OpenTelemetry project. Part of the Cloud Native Computing Foundation, AWS Distro for OpenTelemetry provides open-source APIs, libraries, and agents to collect distributed traces and metrics for application monitoring. With AWS Distro for OpenTelemetry, you can instrument your applications just one time to send correlated metrics and traces to multiple AWS and partner monitoring solutions. Use auto-instrumentation agents to collect traces without changing your code. AWS Distro for OpenTelemetry also collects metadata from your AWS resources and managed services to correlate application performance data with underlying infrastructure data, reducing the mean time to problem resolution. Use AWS Distro for OpenTelemetry to instrument your applications running on Amazon EC2, Amazon ECS, Amazon EKS on Amazon EC2, Fargate, and AWS Lambda, as well as on-premises.

Options for log analysis on AWS

Searching, analyzing, and visualizing log data is an important aspect of understanding distributed systems. Amazon CloudWatch Logs Insights enables you to explore, analyze, and visualize your logs instantly. This allows you to troubleshoot operational problems. Another option for analyzing log files is to use Amazon OpenSearch Service together with Kibana.

Amazon OpenSearch Service can be used for full-text search, structured search, analytics, and all three in combination. Kibana is an open-source data visualization plugin that seamlessly integrates with the Amazon OpenSearch Service.

The following figure demonstrates log analysis with Amazon OpenSearch Service and Kibana. CloudWatch Logs can be configured to stream log entries to Amazon OpenSearch Service in near real time through a CloudWatch Logs subscription. Kibana visualizes the data and exposes a convenient search interface to data stores in Amazon OpenSearch Service. This solution can be used in combination with software like ElastAlert to implement an alerting system to send SNS notifications and emails, create JIRA tickets, and so forth, if anomalies, spikes, or other patterns of interest are detected in the data.



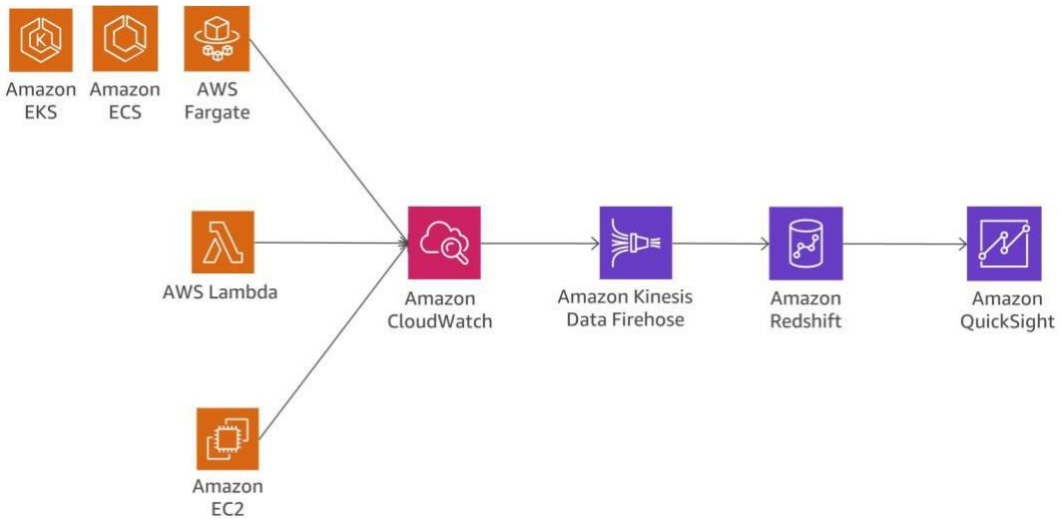
Log analysis with Amazon OpenSearch Service and Kibana

Another option for analyzing log files is to use Amazon Redshift with Amazon QuickSight.

QuickSight can be easily connected to AWS data services, including Redshift, Amazon RDS, Aurora, Amazon EMR,DynamoDB, Amazon S3, and Amazon Kinesis.

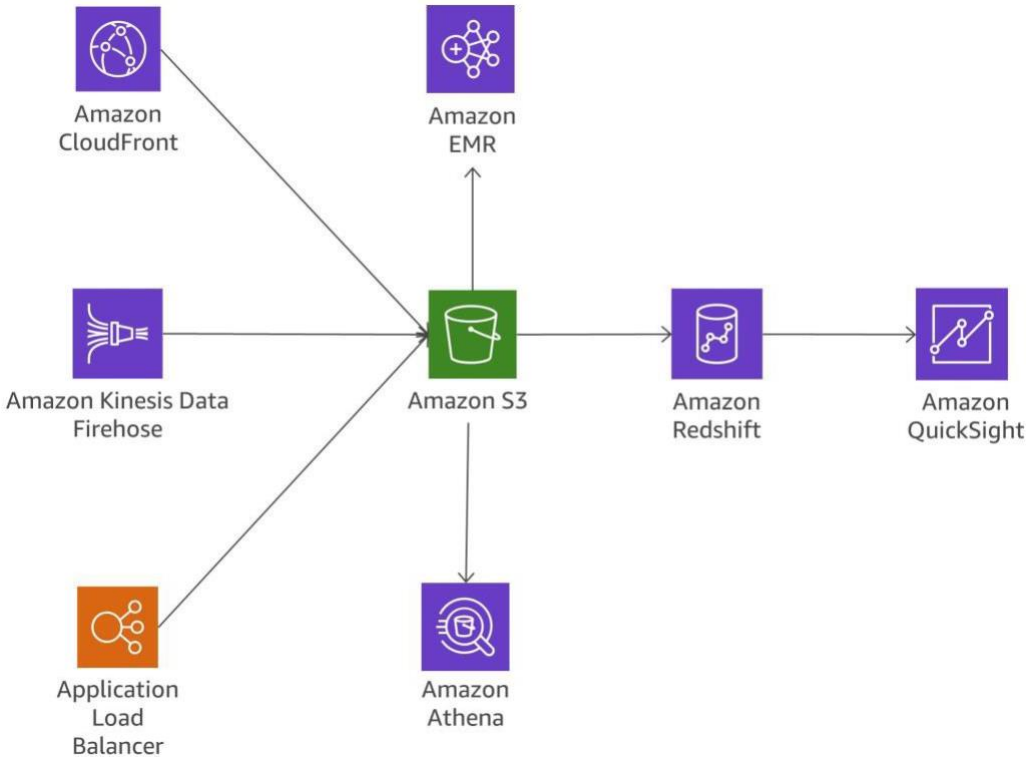
CloudWatch Logs can act as a centralized store for log data, and, in addition to only storing the data, it is possible to stream log entries to Amazon Kinesis Data Firehose.

The following figure depicts a scenario where log entries are streamed from different sources to Redshift using CloudWatch Logs and Kinesis Data Firehose. QuickSight uses the data stored in Redshift for analysis, reporting, and visualization.



Log analysis with Amazon Redshift and Amazon QuickSight

The following figure depicts a scenario of log analysis on Amazon S3. When the logs are stored in Amazon S3 buckets, the log data can be loaded in different AWS data services, such as Redshift or Amazon EMR, to analyze the data stored in the log stream and find anomalies.



Log analysis on Amazon S3

Chattiness

By breaking monolithic applications into small microservices, the communication overhead increases because microservices have to talk to each other. In many implementations, REST over HTTP is used because it is a lightweight communication protocol, but high message volumes can cause issues. In some cases, you might consider consolidating services that send many messages back and forth. If you find yourself in a situation where you consolidate an increased number of services just to reduce chattiness, you should review your problem domains and your domain model.

Protocols

Earlier in this whitepaper, in the section *Asynchronous communication and lightweight messaging*, different possible protocols are discussed. For microservices it is common to use protocols like HTTP. Messages exchanged by services can be encoded in different ways, such as human-readable formats like JSON or YAML, or efficient binary formats such as Avro or Protocol Buffers.

Caching

Caches are a great way to reduce latency and chattiness of microservices architectures. Several caching layers are possible, depending on the actual use case and bottlenecks. Many microservice applications running on AWS use ElastiCache to reduce the volume of calls to other microservices by caching results locally. API Gateway provides a built-in caching layer to reduce the load on the backend servers. In addition, caching is also useful to reduce load from the data persistence layer. The challenge for any caching mechanism is to find the right balance between a good cache hit rate, and the timeliness and consistency of data.

Auditing

Another challenge to address in microservices architectures, which can potentially have hundreds of distributed services, is ensuring visibility of user actions on each service and being able to get a good overall view across all services at an organizational level. To help enforce security policies, it is important to audit both resource access and activities that lead to system changes.

Changes must be tracked at the individual service level as well as across services running on the wider system. Typically, changes occur frequently in microservices architectures, which makes auditing changes even more important. This section examines the key services and features within AWS that can help you audit your microservices architecture.

Audit trail

AWS CloudTrail is a useful tool for tracking changes in microservices because it enables all API calls made in the AWS Cloud to be logged and sent to either CloudWatch Logs in real time, or to Amazon S3 within several minutes.

All user and automated system actions become searchable and can be analyzed for unexpected behavior, company policy violations, or debugging. Information recorded includes a timestamp, user and account information, the service that was called, the service action that was requested, the IP address of the caller, as well as request parameters and response elements.

CloudTrail allows the definition of multiple trails for the same account, which enables different stakeholders, such as security administrators, software developers, or IT

auditors, to create and manage their own trail. If microservice teams have different AWS accounts, it is possible to aggregate trails into a single S3 bucket.

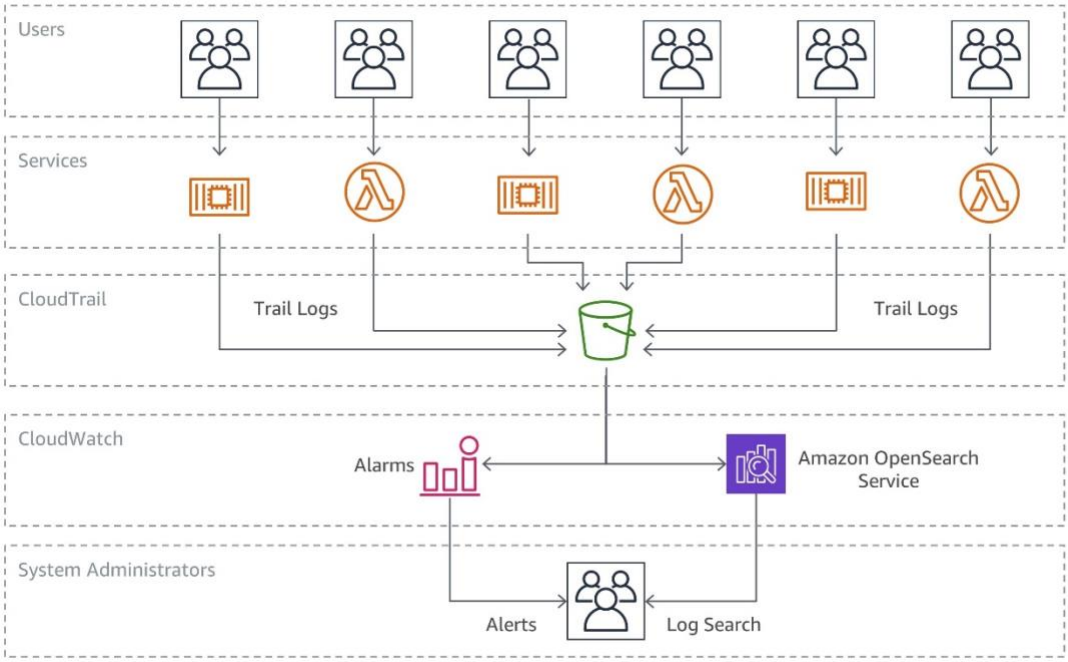
The advantages of storing the audit trails in CloudWatch are that audit trail data is captured in real time, and it is easy to reroute information to Amazon OpenSearch Service for search and visualization. You can configure CloudTrail to log in to both Amazon S3 and CloudWatch Logs.

Events and real-time actions

Certain changes in systems architectures must be responded to quickly and either action taken to remediate the situation, or specific governance procedures to authorize the change must be initiated. The integration of Amazon CloudWatch Events with CloudTrail allows it to generate events for all mutating API calls across all AWS services. It is also possible to define custom events or generate events based on a fixed schedule.

When an event is fired and matches a defined rule, a pre-defined group of people in your organization can be immediately notified, so that they can take the appropriate action. If the required action can be automated, the rule can automatically trigger a built-in workflow or invoke a Lambda function to resolve the issue.

The following figure shows an environment where CloudTrail and CloudWatch Events work together to address auditing and remediation requirements within a microservices architecture. All microservices are being tracked by CloudTrail and the audit trail is stored in an Amazon S3 bucket. CloudWatch Events becomes aware of operational changes as they occur. CloudWatch Events responds to these operational changes and takes corrective action as necessary, by sending messages to respond to the environment, activating functions, making changes, and capturing state information. CloudWatch Events sit on top of CloudTrail and triggers alerts when a specific change is made to your architecture.



Auditing and remediation

Resource inventory and change management

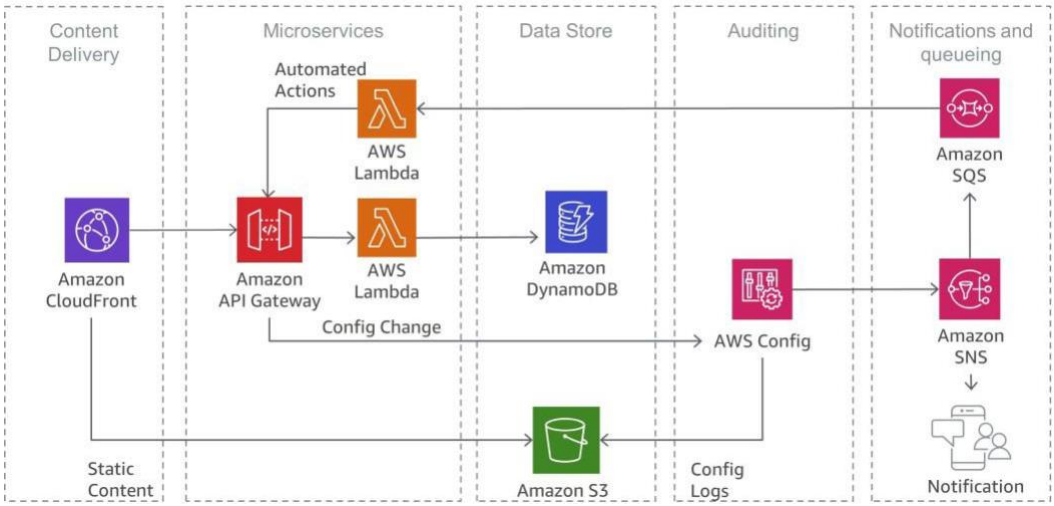
To maintain control over fast-changing infrastructure configurations in an agile development environment, having a more automated, managed approach to auditing and controlling your architecture is essential.

Although CloudTrail and CloudWatch Events are important building blocks to track and respond to infrastructure changes across microservices, AWS Config rules enable a company to define security policies with specific rules to automatically detect, track, and alert you to policy violations.

The next example demonstrates how it is possible to detect, inform, and automatically react to non-compliant configuration changes within your microservices architecture. A member of the development team has made a change to the API Gateway for a microservice to allow the endpoint to accept inbound HTTP traffic, rather than only allowing HTTPS requests.

Because this situation has been previously identified as a security compliance concern by the organization, an AWS Config rule is already monitoring for this condition.

The rule identifies the change as a security violation, and performs two actions: it creates a log of the detected change in an Amazon S3 bucket for auditing, and it creates an SNS notification. Amazon SNS is used for two purposes in our scenario: to send an email to a specified group to inform about the security violation, and to add a message to an SQS queue. Next, the message is picked up, and the compliant state is restored by changing the API Gateway configuration.



Detecting security violations with AWS Config

Resources

- AWS Architecture Center
- AWS Whitepapers
- AWS Architecture Monthly
- AWS Architecture Blog
- This Is My Architecture videos
- AWS Answers
- AWS Documentation

Conclusion

Microservices architecture is a distributed design approach intended to overcome the limitations of traditional monolithic architectures. Microservices help to scale applications and organizations while improving cycle times. However, they also come with a couple of challenges that might add additional architectural complexity and operational burden.

AWS offers a large portfolio of managed services that can help product teams build microservices architectures and minimize architectural and operational complexity. This whitepaper guided you through the relevant AWS services and how to implement typical patterns, such as service discovery or event sourcing, natively with AWS services.

Document Revisions

Date	Description
November 9, 2021	Integration of Amazon EventBridge, AWS OpenTelemetry, AMP, AMG, Container Insights, minor text changes.
August 1, 2019	Minor text changes.
June 1, 2019	Integration of Amazon EKS, AWS Fargate, Amazon MQ,AWS PrivateLink, AWS App Mesh, AWS Cloud Map
September 1, 2017	Integration of AWS Step Functions, AWS X-Ray, and ECSevent streams.
December 1, 2016	First publication

Contributors

The following individuals contributed to this document:

- Sascha Möllering, Solutions Architecture, AWS
- Christian Müller, Solutions Architecture, AWS
- Matthias Jung, Solutions Architecture, AWS
- Peter Dalbhanjan, Solutions Architecture, AWS
- Peter Chapman, Solutions Architecture, AWS
- Christoph Kassen, Solutions Architecture, AWS

- Umair Ishaq, Solutions Architecture, AWS
- Rajiv Kumar, Solutions Architecture, AWS