# Implementing Microservices on AWS

*June 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.

# Contents

# Abstract

Microservices are an architectural and organizational approach to software development 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 to work independently from each other. Using a microservices approach, software is composed of small services that communicate over well-defined APIs that can be deployed independently. These services are owned by small autonomous teams. This agile approach is key to successfully scale your organization.

There are three common patterns that we observe when our customers build microservices: **API driven**, **event driven**, and **data streaming**. In this whitepaper, we introduce all three approaches and summarize the common characteristics of microservices, discuss the main challenges of building microservices, and describe how product teams can leverage Amazon Web Services (AWS) to overcome these challenges.

# 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 leveraged for microservices.[1]

We first describe different aspects of a highly scalable, fault-tolerant microservices architecture (user interface, microservices implementation, and data store) and how to build it on AWS leveraging container technologies. We then recommend the AWS services for implementing a typical serverless microservices architecture in order 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, we look at the overall system and discuss the cross-service aspects of a microservices architecture, such as distributed monitoring and auditing, data consistency, and asynchronous communication.

# Simple 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. Figure 1 depicts a reference architecture for a typical microservices application on AWS.

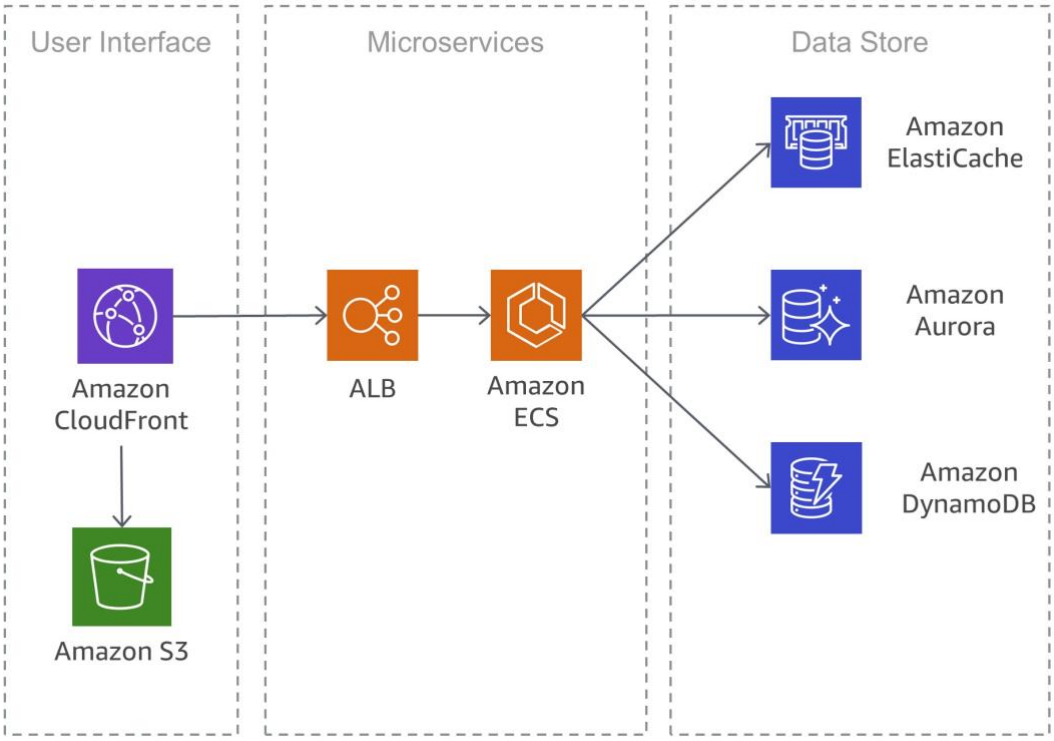


*Figure 1: Typical microservices application on AWS*

# 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[2]) and Amazon CloudFront[3].

Since 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 CDN. 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.

# Microservices

We often say that APIs are the front door of microservices. By that, we mean that APIs serve as the entry point for applications logic behind a set of programmatic interfaces, typically a RESTful web services API.[4] This API accepts and processes calls from clients and might implement functionality such as traffic management, request filtering, routing, caching, authentication, and authorization.

## Microservices Implementations

AWS has integrated building blocks that support the development of microservices. Two popular approaches are using AWS Lambda[5] and Docker containers with AWS Fargate[6].

With AWS Lambda, you simply upload your code and let Lambda take care of everything required to run and scale the execution to meet your actual demand curve with high availability. This means, there is no administration of infrastructure needed. Lambda supports several programming languages and can be triggered 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[7] 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[8]) and Amazon Elastic Kubernetes Service (Amazon EKS[9]) eliminate the need to install, operate, and scale your own cluster management infrastructure. With simple 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[10]) volumes, and AWS Identity and Access Management (IAM)[11] roles.

AWS Fargate is a container management service that allows you to run serverless containers so you don't have worry about provisioning, configuring, and scaling clusters of virtual machines to run containers. 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 terminates tasks. A task placement constraint is a rule that is considered during task placement. You can associate attributes, 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 the latest version 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 masters. The IAM integration allows you to use IAM to directly authenticate with the master itself as provide fine granular access to the public endpoint of your Kubernetes masters.

Docker images used in Amazon ECS and Amazon EKS can be stored in Amazon Elastic Container Registry (Amazon ECR.)[12] Amazon ECR eliminates the need to operate and scale the infrastructure required to power your container registry.

Continuous integration and continuous delivery (CI/CD) is a best practice and a vital part of a DevOps initiative that enables rapid software changes while maintaining system stability and security. However, this is out of the scope of this whitepaper, more information can be found in the “Practicing Continuous Integration and Continuous Delivery on AWS” whitepaper[13].

## Private Links

AWS PrivateLink[14] is a highly available, scalable technology that enables you to privately connect your 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, NAT 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 of microservices architectures, e.g., it is possible to create hundreds of VPCs, each hosting and providing a single microservice. Companies can now create services and offer them for sale to other AWS customers, for access via a private connection. They create a service that accepts TCP traffic, host it behind a Network Load Balancer, and then make the service available, either directly or in AWS Marketplace. They will be notified of new subscription requests and can choose to accept or reject each one. While the power of AWS PrivateLink has merits in any number of scenarios, it's of particular interest to SaaS organizations. Through AWS PrivateLink, SaaS providers see new and creative opportunities to use this networking construct to enhance and expand the architectural and business models of their solutions.

## 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[15] 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 allow resources to be used to support more writes. Caches also can 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[16]) as managed services via Amazon Relational Database Service (Amazon RDS[17]).

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[18] 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. DynamoDB Accelerator (DAX)[19] 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, DynamoDB provides 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 we have described is already using managed services, but we still have to manage Amazon Elastic Compute Cloud (Amazon EC2[20]) instances. We can further reduce the operational efforts needed to run, maintain, and monitor microservices 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 (i.e., 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 protect the backend services, caching API responses, handling request and response transformation, and generating API definitions and documentation with tools such as Swagger.[21]

Amazon API Gateway[22] 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.

Figure 2 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 (PoP) to minimize latency and provide optimum user experience.

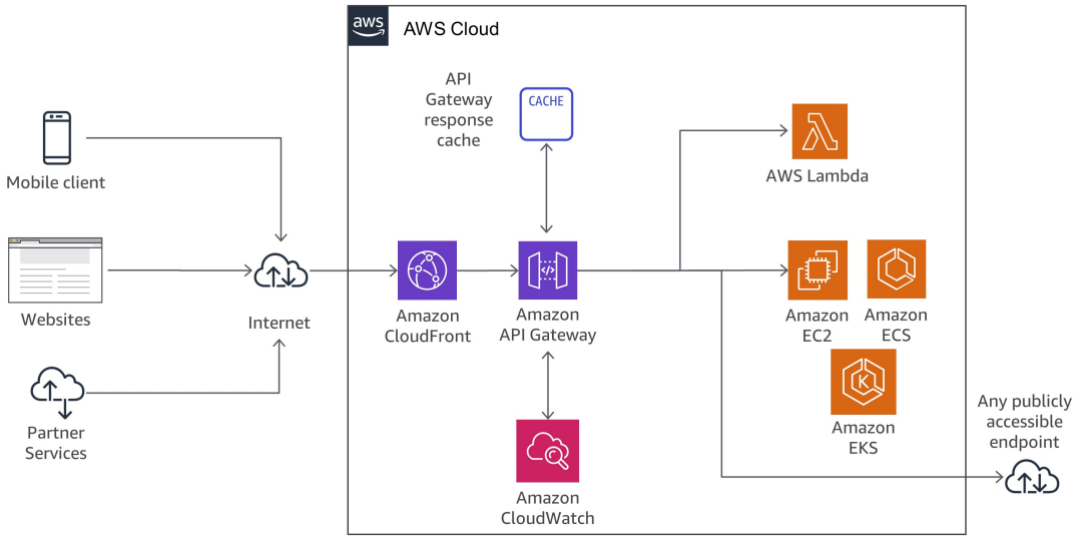


*Figure 2: API Gateway call flow*

# Serverless Microservices

*"No server is easier to manage than no server".*[23] Getting rid of servers is a great way to eliminate operational complexity.

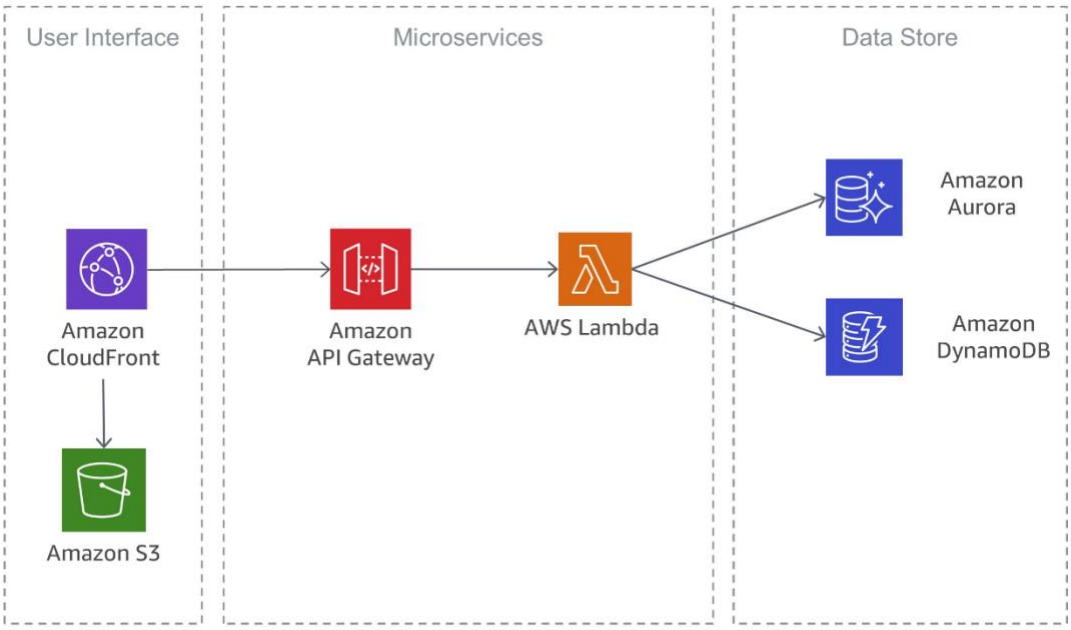


*Figure 3: Serverless microservice using AWS Lambda*

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 our documentation.[24]

Figure 3 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.

Figure 4 shows a similar implementation that is also based on serverless services. In this architecture, Docker containers are used with AWS Fargate, so it's not necessary to care about the underlying infrastructure. In addition to Amazon DynamoDB, Amazon Aurora Serverless[25] is used, which is an on-demand, auto-scaling configuration for Amazon 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.

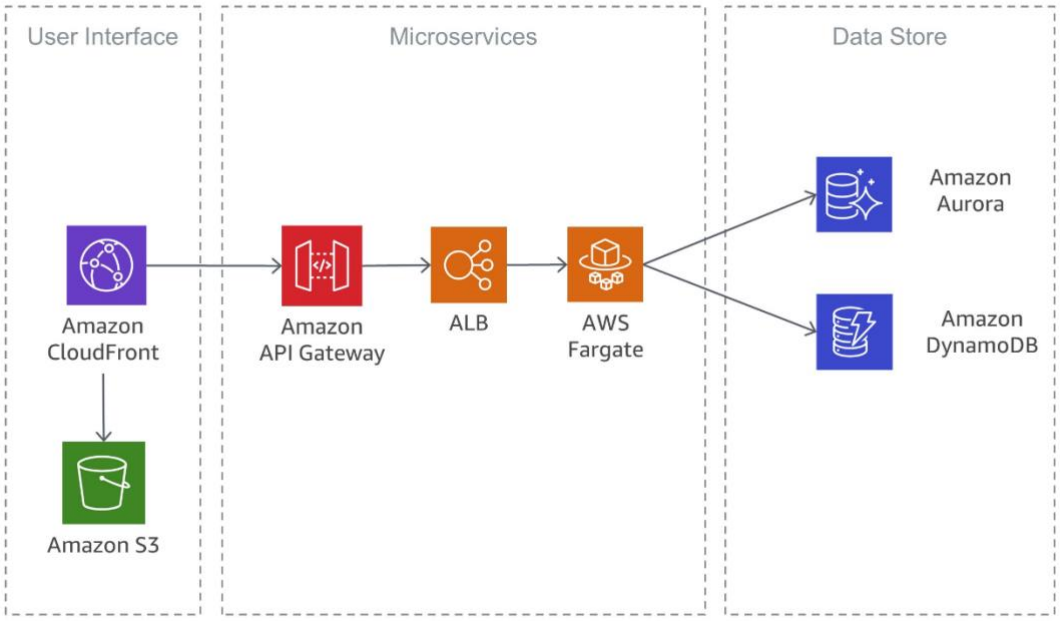


*Figure 4: Serverless microservice using AWS Fargate*

# Deploying Lambda-Based Applications

You can use AWS CloudFormation[26] to define, deploy, and configure serverless applications.

The AWS Serverless Application Model (AWS SAM) is a convenient way to define serverless applications.[27] AWS SAM is natively supported by CloudFormation and defines a simplified syntax for expressing serverless resources. In order to deploy your application, simply 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 CLI 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, we now want to focus on cross-service challenges, such as service discovery, data consistency, asynchronous communication, and distributed monitoring and auditing.

## Service Discovery

One of the primary challenges with microservices architectures is allowing services to discover and interact with each other. The distributed characteristics of microservices 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-store information, such as configuration data, that can be used by applications. In this section, we explore several techniques for performing service discovery on AWS for microservices-based architectures.

### DNS-Based Service Discovery

Amazon ECS now includes integrated service discovery that makes it easy for 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[28], AWS Lambda, and ECS Event Stream, 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 leverage unified service discovery for services managed by Kubernetes. To enable this integration, AWS contributed to the External DNS project[29], a Kubernetes incubator project.

Another option is to leverage the capabilities of AWS Cloud Map[30]. AWS Cloud Map extends the capabilities of the Auto Naming APIs by providing a service registry for resources, such as IPs, URLs, and 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-party software like HashiCorp Consul,[31] etcd,[32] or Netflix Heureka.[33] All three examples are distributed, reliable key-value stores. For HashiCorp Consul, there is an AWS Quick Start[34] 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 inter-service communication, which is responsible for monitoring and controlling traffic in microservice architectures. This allows 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 for communicating 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[35] is a service mesh that provides application-level networking to make it easy for your services to communicate with each other across multiple types of compute infrastructure. App Mesh standardizes how your services communicate, giving you end-to-end visibility and ensuring high availability for your applications.

You can use AWS App Mesh with existing or new microservices running on AWS 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,[36] distributed microservices 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 leverage a single ACID[37] 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[38] make it easy to implement a Saga execution coordinator as shown in the next figure.

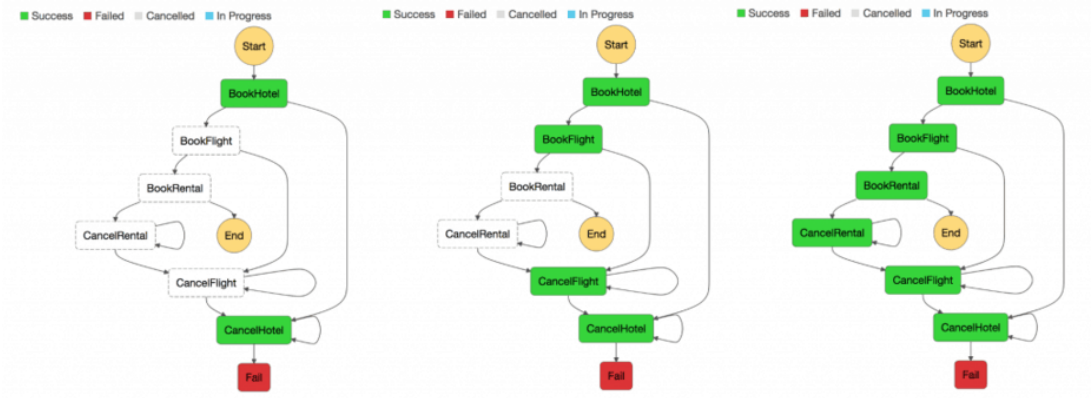


*Figure 5: Saga execution coordinator*

Building a centralized store of critical reference data that is curated by master data management tools and procedures provides a means for microservices to synchronize their critical data and possibly roll back state.[39] Using Lambda with scheduled Amazon CloudWatch Events you can build a simple cleanup and deduplication mechanism.[40]

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.[41] 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/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 CQRS (Command Query Responsibility Segregation) pattern to decouple read from write workloads and optimize both for performance, scalability, and security.[42] In traditional data management systems, commands and queries are run against the same data repository.

Figure 6 shows how the event sourcing pattern can be implemented on AWS. Amazon Kinesis Data Streams[43] serves as the main component of the central event store, which captures application changes as events and persists them on Amazon S3.

Figure 6 depicts three different microservices composed of Amazon API Gateway, AWS Lambda, and Amazon DynamoDB. The blue 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.

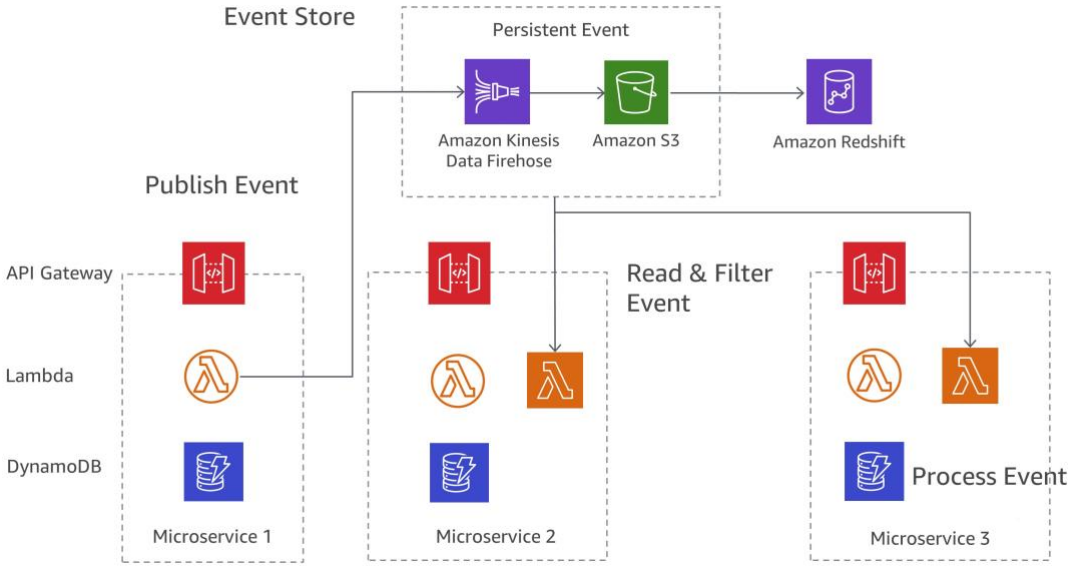


*Figure 6: Event sourcing pattern on AWS*

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.

# 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, such as workloads running on Amazon EC2 and Amazon ECS, code running on Lambda, or any web application. 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, 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

An additional pattern to implement communication between microservices is message passing. Services communicate by exchanging messages via a queue. One major benefit of this communication style is that it's not necessary to have a service discovery and services are loosely couple. 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[44]) and Amazon Simple Notification Service (Amazon SNS[45]).

Both services work closely together: Amazon SNS allows 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. Figure 7 demonstrates the integration of Amazon SNS and Amazon SQS.

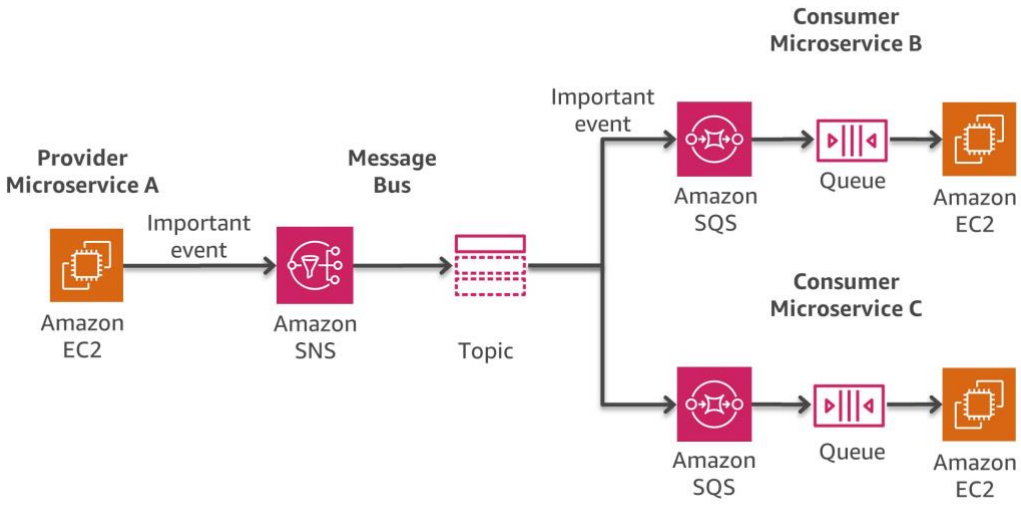


*Figure 7: Message bus pattern on AWS*

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.

A different implementation strategy is based on Amazon MQ[46], 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 e.g. 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 as it introduces tighter coupling and makes it harder to quickly replace individual services.

You can use 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 and serialization/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 simple to build and run distributed services. Step Functions automatically triggers 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 and Amazon ECS, and additional services like Amazon SageMaker[47] and AWS Glue[48]. Figure 8 illustrates how invocations of Lambda functions are pushed directly from Step Functions to Lambda, whereas workers on Amazon EC2 or Amazon ECS continuously poll for invocations.

Step Functions manages the operations and underlying infrastructure for you to help ensure that your application is available at any scale.

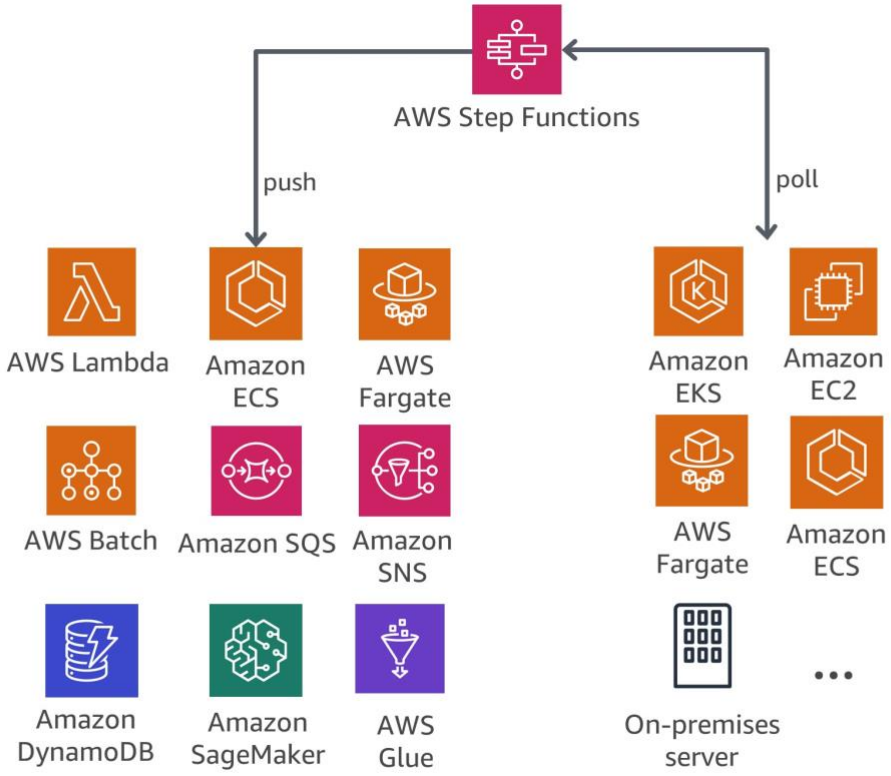


*Figure 8: Orchestration with AWS Step Functions*

To build workflows, Step Functions uses the Amazon States Language.[49] Workflows can contain sequential or parallel steps as well as branching steps.

Figure 9 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.

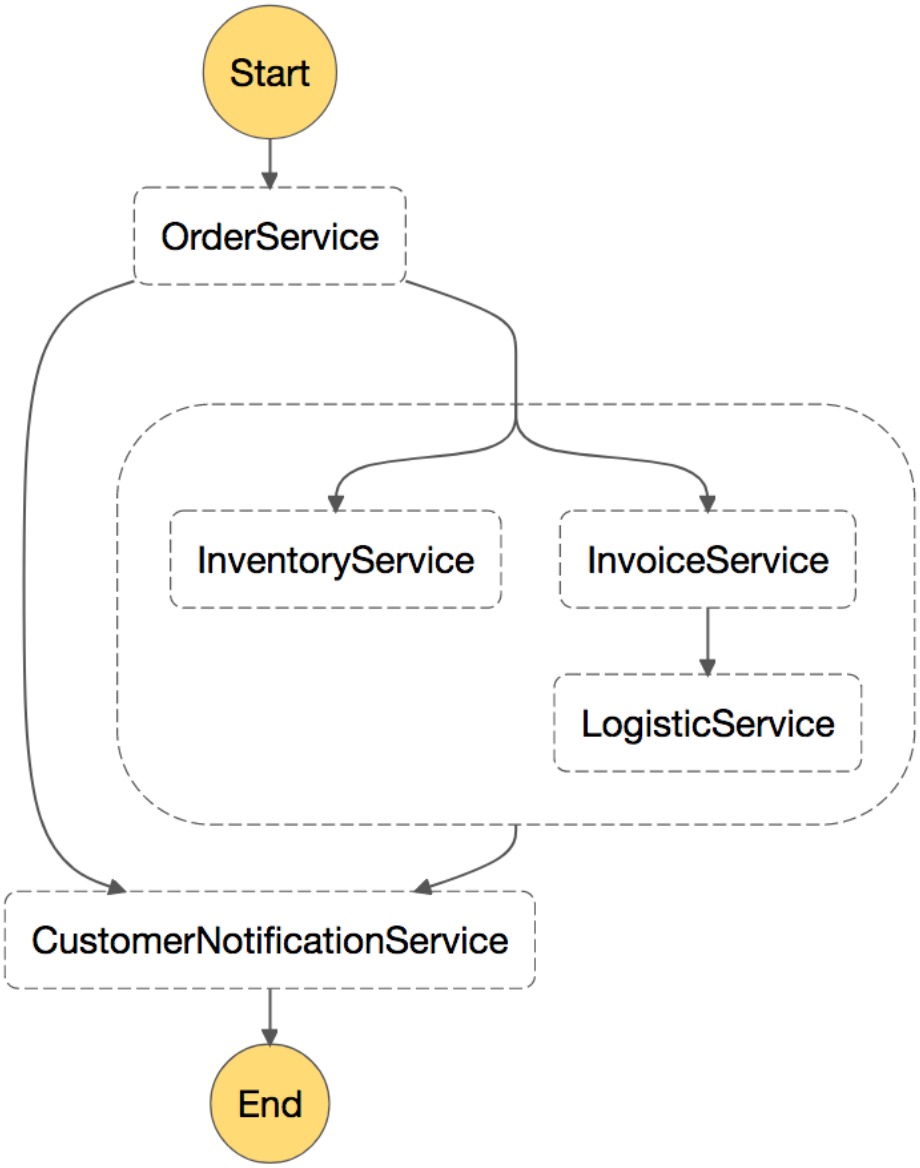


*Figure 9: An example of a microservices workflow invoked by AWS Step Functions*

## Distributed Monitoring

A microservices architecture consists of many different distributed parts that have to be monitored.

You can use Amazon CloudWatch[50] 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 EC2 instances, DynamoDB tables, and 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 to that, dynamic scaling can be implemented based on custom metrics.[51]

Another popular option–especially for Amazon EKS–is to use Prometheus[52]. Prometheus is an open-source monitoring and alerting toolkit that is often used in combination with Grafana [53]to visualize the collected metrics. Many Kubernetes components store metrics at /metrics and Prometheus can scrape these metrics at a regular interval.

## Centralizing Logs

Consistent logging is critical for troubleshooting and identifying issues. Microservices allow 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 improving an application gradually.

Most AWS services centralize their log files by default. The primary destinations for log files on AWS are Amazon S3 and Amazon CloudWatch Logs. For applications running on EC2 instances, a daemon is available to send log files to CloudWatch Logs. Lambda functions natively send their log output to CloudWatch Logs and Amazon ECS includes support for the `awslogs` log driver that allows the centralization of container logs to CloudWatch Logs.[54] For Amazon EKS, it is necessary to run FluentD which forwards logs from the individual instances in the cluster to a centralized logging CloudWatch Logs where they are combined for higher-level reporting using Elasticsearch and Kibana.

Figure 10 illustrates the logging capabilities of some of the services. Teams are then able to search and analyze these logs using tools like Amazon Elasticsearch Service (Amazon ES)[55] and Kibana. Amazon Athena[56] can be used to run ad hoc queries against centralized log files in Amazon S3.

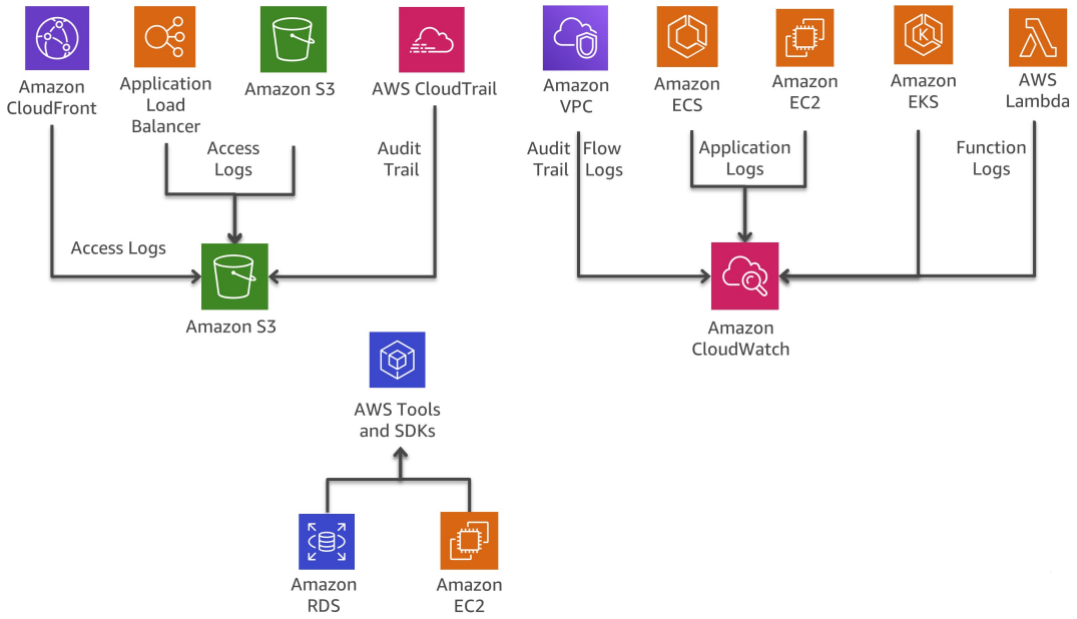


*Figure 10: Logging capabilities of AWS services*

## 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 behind AWS X-Ray[57] 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. Via the X-Ray SDK, any microservice can read but can also add or update this header.

AWS X-Ray works with Amazon EC2, Amazon ECS, Lambda, and AWS Elastic Beanstalk[58]. You can use X-Ray with applications written in Java, Node.js, and .NET that are deployed on these services.

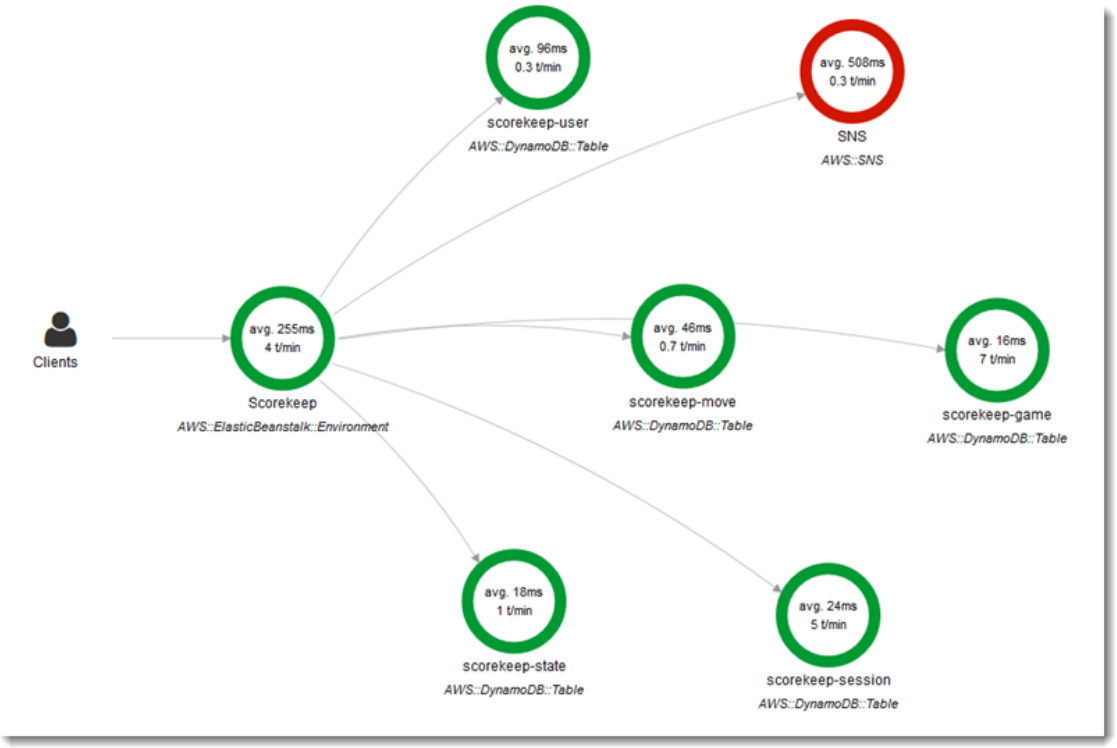


*Figure 11: AWS X-Ray service map*

## Options for Log Analysis on AWS

Searching, analyzing, and visualizing log data is an important aspect of understanding distributed systems. Amazon CloudWatch Logs Insights is a great service 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 ES together with Kibana.

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

Figure 12 demonstrates log analysis with Amazon ES and Kibana. CloudWatch Logs can be configured to stream log entries to Amazon ES in near real time through a CloudWatch Logs subscription. Kibana visualizes the data and exposes a convenient search interface to data stores in Amazon ES. This solution can be used in combination with software like ElastAlert to implement an alerting system in order to send SNS notifications, emails, create JIRA tickets, etc., if anomalies, spikes, or other patterns of interest are detected in the data.[59]

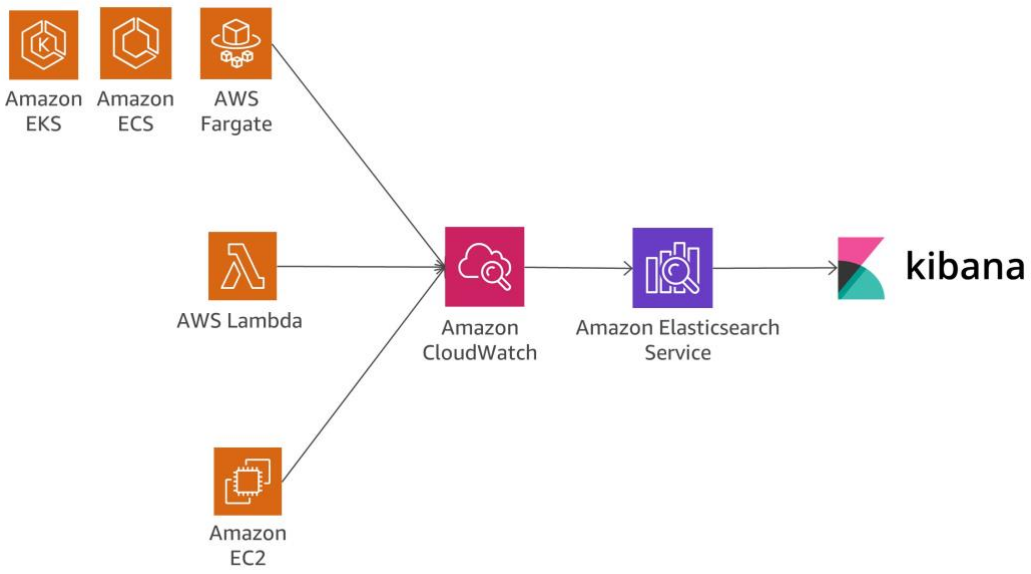


*Figure 12: Log analysis with Amazon Elasticsearch Service and Kibana*

Another option for analyzing log files is to use Amazon Redshift[60] together with Amazon QuickSight[61].

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

Amazon 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.

Figure 13 depicts a scenario where log entries are streamed from different sources to Amazon Redshift using CloudWatch Logs and Kinesis Data Firehose. Amazon QuickSight uses the data stored in Amazon Redshift for analysis, reporting, and visualization.

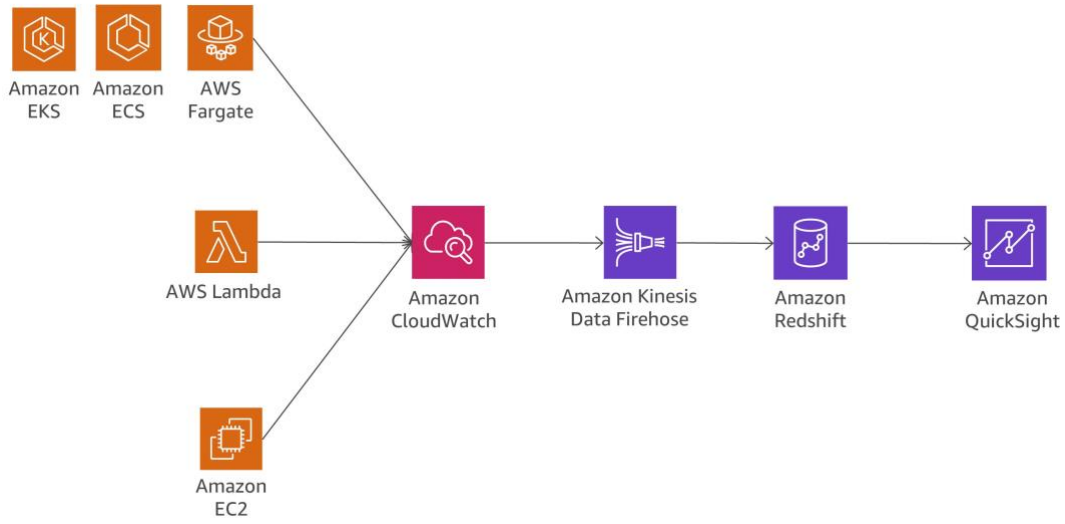


*Figure 13: Log analysis with Amazon Redshift and Amazon QuickSight*

Figure 14 depicts a scenario of log analysis on Amazon S3. When the logs are stored in S3 buckets, the log data can be loaded in different AWS data services, such as Amazon Redshift or Amazon EMR, to analyze the data stored in the log stream and find anomalies.

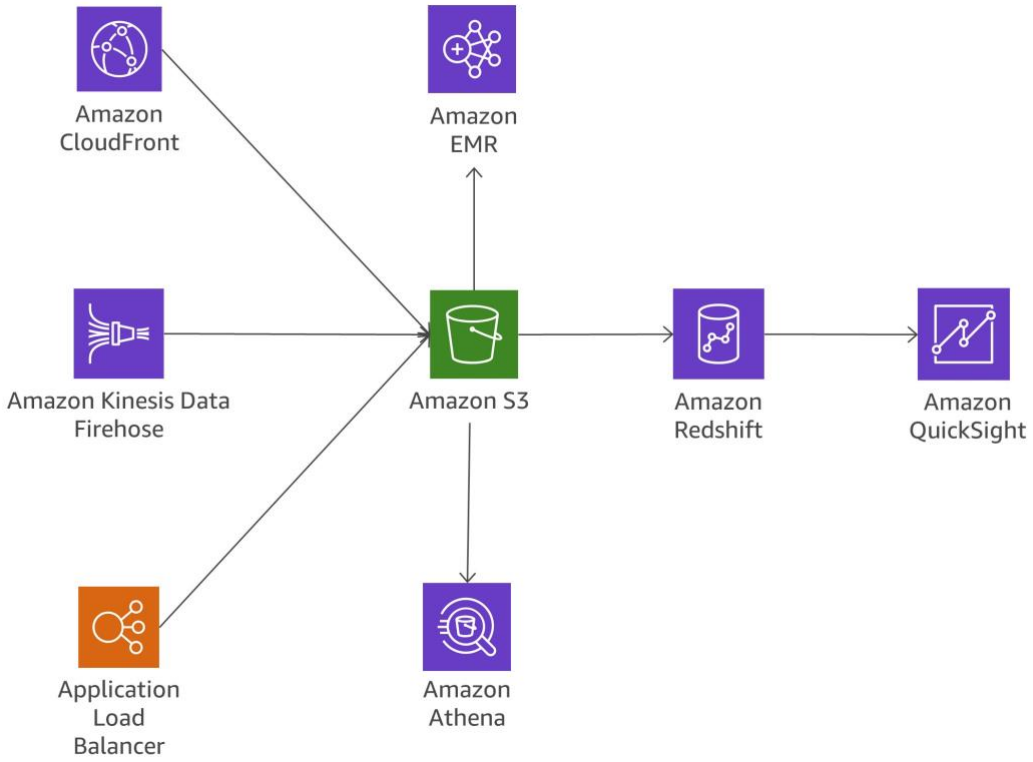


*Figure 14: Log analysis on Amazon S3*

# 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 more and more of your 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 simple 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 Amazon 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/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 as well as 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. In this section, we look at the key services and features within AWS that can help you audit your microservices architecture.

## Audit Trail

AWS CloudTrail[62] 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/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 allows 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.[63]

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 ES for search and visualization. You can configure CloudTrail to log into 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 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, the right people in your organization can be immediately notified, enabling them to 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.

Figure 15 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 S3 bucket. CloudWatch Events sit on top of CloudTrail and triggers alerts when a specific change is made to your architecture.

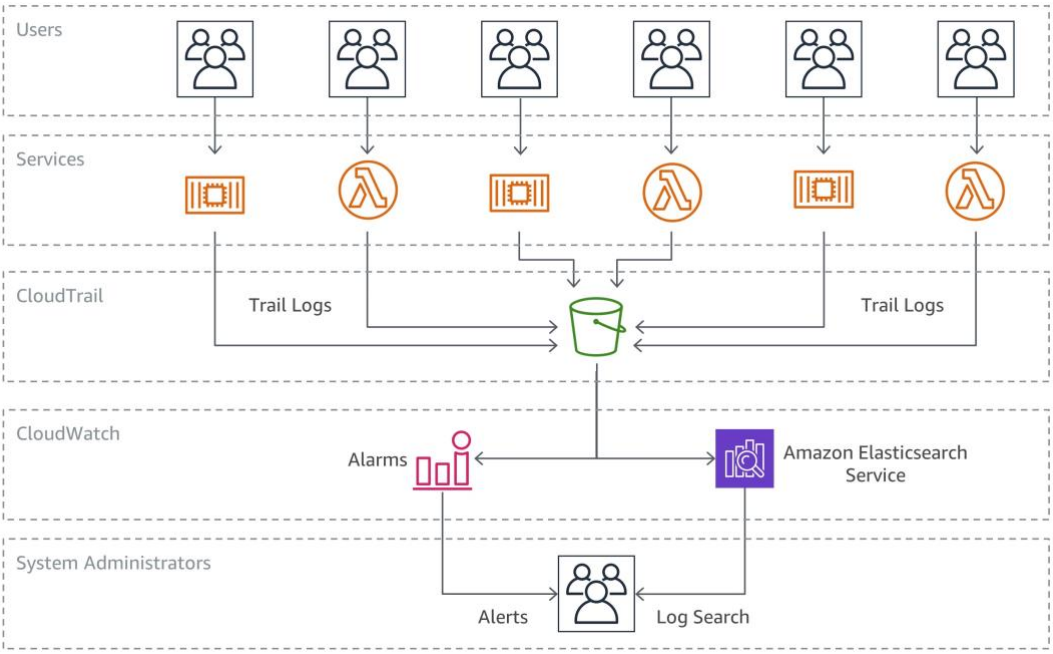


*Figure 15: Auditing and remediation*

## 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.

While CloudTrail and CloudWatch Events are important building blocks to track and respond to infrastructure changes across microservices, AWS Config[64] rules allow 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 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.

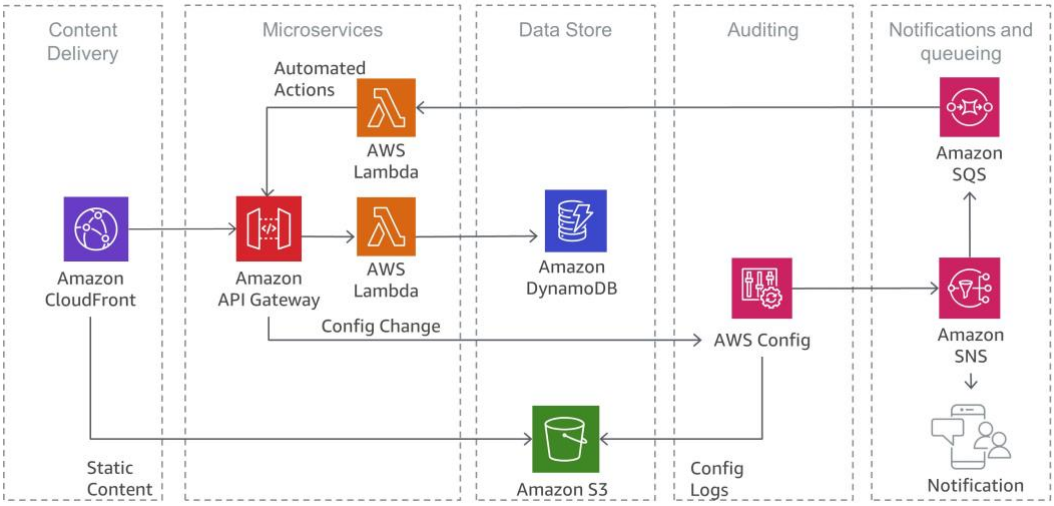


*Figure 16: Detecting security violations with AWS Config*

# 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 guides you through the relevant AWS services and how to implement typical patterns, such as service discovery or event sourcing, natively with AWS services.

# Contributors

The following individuals and organizations 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

# Document Revisions

| Date | Description |
|---|---|
| **June 2019** | Integration of Amazon EKS, AWS Fargate, Amazon MQ, AWS PrivateLink, AWS App Mesh, AWS Cloud Map |
| **September 2017** | Integration of AWS Step Functions, AWS X-Ray, and ECS event streams. |
| **December 2016** | First publication |

# Notes

[1] https://12factor.net/

[2] https://aws.amazon.com/s3/

[3] https://aws.amazon.com/cloudfront/

[4] https://en.wikipedia.org/wiki/Representational_state_transfer

[5] https://aws.amazon.com/lambda/

[6] https://aws.amazon.com/fargate/

[7] https://www.docker.com/

[8] https://aws.amazon.com/ecs/

[9] https://aws.amazon.com/eks/

[10] https://aws.amazon.com/ebs/

[11] https://aws.amazon.com/iam/

[12] https://aws.amazon.com/ecr/

[13] https://d1.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf

[14] https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpce-interface.html

[15] https://aws.amazon.com/elasticache/

[16] https://aws.amazon.com/rds/aurora/

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

[18] https://aws.amazon.com/dynamodb/

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

[20] https://aws.amazon.com/ec2/

[21] http://swagger.io/

[22] https://aws.amazon.com/api-gateway/

[23] https://twitter.com/awsreinvent/status/652159288949866496

[24] http://docs.aws.amazon.com/apigateway/latest/developerguide/getting-started.html

[25] https://aws.amazon.com/rds/aurora/serverless/

[26] https://aws.amazon.com/cloudformation/

[27] https://github.com/awslabs/serverless-application-model

[28] https://aws.amazon.com/route53/

[29] https://github.com/kubernetes-incubator/external-dns

[30] https://aws.amazon.com/cloud-map/

[31] https://www.consul.io/

[32] https://github.com/coreos/etcd

[33] https://github.com/Netflix/eureka

[34] https://aws.amazon.com/quickstart/architecture/consul/

[35] https://aws.amazon.com/app-mesh/

[36] https://en.wikipedia.org/wiki/CAP_theorem

[37] https://en.wikipedia.org/wiki/ACID_(computer_science)

[38] https://aws.amazon.com/step-functions/

[39] https://en.wikipedia.org/wiki/Master_data_management

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

[41] http://martinfowler.com/eaaDev/EventSourcing.html

[42] http://martinfowler.com/bliki/CQRS.html

[43] https://aws.amazon.com/kinesis/data-streams/
[44] https://aws.amazon.com/sqs/
[45] https://aws.amazon.com/sns/
[46] https://aws.amazon.com/amazon-mq/
[47] https://aws.amazon.com/sagemaker/
[48] https://aws.amazon.com/glue/

[49] https://states-language.net/spec.html

[50] https://aws.amazon.com/cloudwatch/

[51] https://docs.aws.amazon.com/autoscaling/latest/userguide/policy_creating.html

[52] https://prometheus.io/docs/introduction/overview/

[53] https://grafana.com/

[54] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/using_awslogs.html

[55] https://aws.amazon.com/elasticsearch-service/
[56] https://aws.amazon.com/athena/
[57] https://aws.amazon.com/xray/
[58] https://aws.amazon.com/elasticbeanstalk/

[59] https://github.com/Yelp/elastalert

[60] https://aws.amazon.com/redshift/
[61] https://aws.amazon.com/quicksight/
[62] https://aws.amazon.com/cloudtrail/

[63] http://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-receive-logs-from-multiple-accounts.html

[64] https://aws.amazon.com/config/