AWS Whitepaper

Implementing Microservices on AWS

Implementing Microservices on AWS: AWS Whitepaper

Table of Contents

Implementing Microservices on AWS

Publication date: **July 31, 2023** (*Document history*)

Microservices offer a streamlined approach to software development that accelerates deployment, encourages innovation, enhances maintainability, and boosts scalability. This method relies on small, loosely coupled services that communicate through well-defined APIs, which are managed by autonomous teams. Adopting microservices offers benefits, such as improved scalability, resilience, flexibility, and faster development cycles.

This whitepaper explores three popular microservices patterns: API driven, event driven, and data streaming. We provide an overview of each approach, outline microservices' key features, address the challenges in their development, and illustrate how Amazon Web Services (AWS) can help application teams tackle these obstacles.

Considering the complex nature of topics like data store, asynchronous communication, and service discovery, you are encouraged to weigh your application's specific needs and use cases alongside the guidance provided when making architectural decisions.

Introduction

Microservices architectures combine successful and proven concepts from various fields, such as:

- Agile software development
- Service-oriented architectures
- API-first design
- Continuous Integration/Continuous Delivery (CI/CD)

Often, microservices incorporate design patterns from the Twelve-Factor App.

While microservices offer many benefits, it's vital to assess your use case's unique requirements and associated costs. Monolithic architecture or alternative approaches may be more appropriate in some cases. Deciding between microservices or monoliths should be made on a case-by-case basis, considering factors like scale, complexity, and specific use cases.

We first explore a highly scalable, fault-tolerant microservices architecture (user interface, microservices implementation, data store) and demonstrate how to build it on AWS using container

technologies. We then suggest AWS services to implement a typical serverless microservices architecture, reducing operational complexity.

Serverless is characterized by the following principles:

- No infrastructure to provision or manage
- Automatically scaling by unit of consumption
- "Pay for value" billing model
- Built-in availability and fault tolerance
- Event Driven Architecture (EDA)

Lastly, we examine the overall system and discuss cross-service aspects of a microservices architecture, such as distributed monitoring, logging, tracing, auditing, data consistency, and asynchronous communication.

This document focuses on workloads running in the AWS Cloud, excluding hybrid scenarios and migration strategies. For information on migration strategies, refer to the Container Migration Methodology whitepaper.

Are you Well-Architected?

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems. Using the AWS Well-Architected Tool, available at no charge in the AWS Management Console, you can review your workloads against these best practices by answering a set of questions for each pillar.

In the Serverless Application Lens, we focus on best practices for architecting your serverless applications on AWS.

For more expert guidance and best practices for your cloud architecture—reference architecture deployments, diagrams, and whitepapers—refer to the AWS Architecture Center.

Modernizing to microservices

Microservices are essentially small, independent units that make up an application. Transitioning from traditional monolithic structures to microservices can follow various strategies.

This transition also impacts the way your organization operates:

- It encourages agile development, where teams work in quick cycles.
- Teams are typically small, sometimes described as *two pizza teams*—small enough that two pizzas could feed the entire team.
- Teams take full responsibility for their services, from creation to deployment and maintenance.

Simple microservices architecture on AWS

Typical monolithic applications consist of different layers: a presentation layer, an application layer, and a data layer. Microservices architectures, on the other hand, separate functionalities into cohesive *verticals* according to specific domains, rather than technological layers. Figure 1 illustrates a reference architecture for a typical microservices application on AWS.

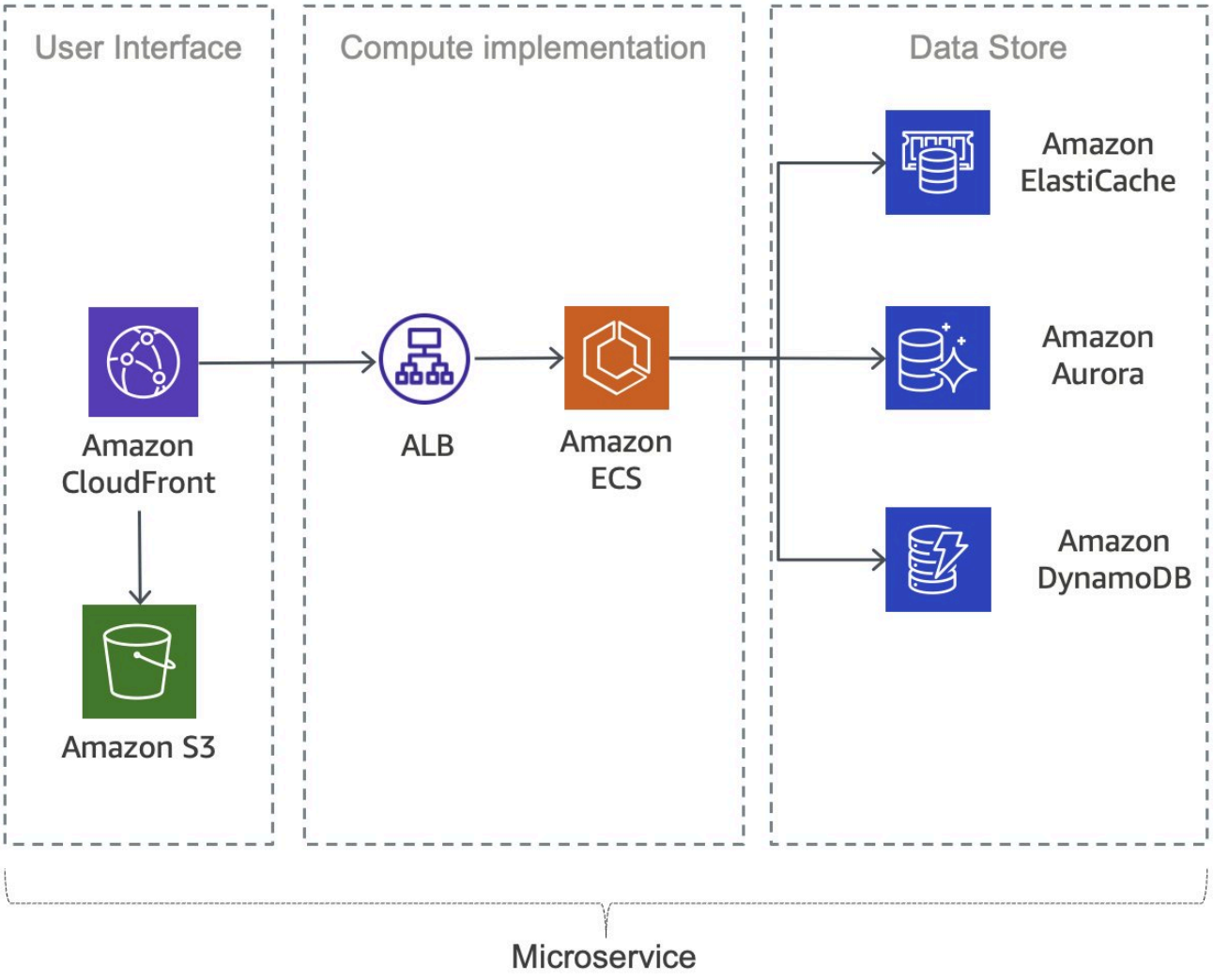

Figure 1: Typical microservices application on AWS

User interface

Modern web applications often use JavaScript frameworks to develop single-page applications that communicate with backend APIs. These APIs are typically built using Representational State

Transfer (REST) or RESTful APIs, or GraphQL APIs. Static web content can be served using Amazon Simple Storage Service (Amazon S3) and Amazon CloudFront.

Microservices

APIs are considered the *front door* of microservices, as they are the entry point for application logic. Typically, RESTful web services API or GraphQL APIs are used. These APIs manage and process client calls, handling functions such as traffic management, request filtering, routing, caching, authentication, and authorization.

Microservices implementations

AWS offers building blocks to develop microservices, including Amazon ECS and Amazon EKS as the choices for container orchestration engines and AWS Fargate and EC2 as hosting options. AWS Lambda is another serverless way to build microservices on AWS. Choice between these hosting options depends on the customer's requirements to manage the underlying infrastructure.

AWS Lambda allows you to upload your code, automatically scaling and managing its execution with high availability. This eliminates the need for infrastructure management, so you can move quickly and focus on your business logic. Lambda supports multiple programming languages and can be triggered by other AWS services or called directly from web or mobile applications.

Container-based applications have gained popularity due to portability, productivity, and efficiency. AWS offers several services to build, deploy and manage containers.

- App2Container, a command line tool for migrating and modernizing Java and .NET web applications into container format. AWS A2C analyzes and builds an inventory of applications running in bare metal, virtual machines, Amazon Elastic Compute Cloud (EC2) instances, or in the cloud.

- Amazon Elastic Container Service (Amazon ECS) and Amazon Elastic Kubernetes Service (Amazon EKS) manage your container infrastructure, making it easier to launch and maintain containerized applications.
  - Amazon EKS is a managed Kubernetes service to run Kubernetes in the AWS cloud and on-premises data centers (Amazon EKS Anywhere). This extends cloud services into on-premises environments for low-latency, local data processing, high data transfer costs, or data residency requirements (see the whitepaper on "Running Hybrid Container Workloads With Amazon EKS

Anywhere"). You can use all the existing plug-ins and tooling from the Kubernetes community with EKS.

- Amazon Elastic Container Service (Amazon ECS) is a fully managed container orchestration service that simplifies your deployment, management, and scaling of containerized applications. Customers choose ECS for simplicity and deep integration with AWS services.

For further reading, see the blog Amazon ECS vs Amazon EKS: making sense of AWS container services.

- AWS App Runner is a fully managed container application service that lets you build, deploy, and run containerized web applications and API services without prior infrastructure or container experience.
- AWS Fargate, a serverless compute engine, works with both Amazon ECS and Amazon EKS to automatically manage compute resources for container applications.
- Amazon ECR is a fully managed container registry offering high-performance hosting, so you can reliably deploy application images and artifacts anywhere.

Continuous integration and continuous deployment (CI/CD)

Continuous integration and continuous delivery (CI/CD) is a crucial part of a DevOps initiative for rapid software changes. AWS offers services to implement CI/CD for microservices, but a detailed discussion is beyond the scope of this document. For more information, see the Practicing Continuous Integration and Continuous Delivery on AWS whitepaper.

Private networking

AWS PrivateLink is a technology that enhances the security of microservices by allowing private connections between your Virtual Private Cloud (VPC) and supported AWS services. It helps isolate and secure microservices traffic, ensuring it never crosses the public internet. This is particularly useful for complying with regulations like PCI or HIPAA.

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 allow 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's 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. 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, 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.

For more information, see Distributed data management and How to Choose a Database.

Simplifying operations

To further simplify the operational efforts needed to run, maintain, and monitor microservices, we can use a fully serverless architecture.

Topics

Deploying Lambda-based applications

You can deploy your Lambda code by uploading a `zip` file archive or by creating and uploading a container image through the console UI using a valid Amazon ECR image URI. However, when a Lambda function becomes complex, meaning it has layers, dependencies, and permissions, uploading through the UI can become unwieldy for code changes.

Using AWS CloudFormation and the AWS Serverless Application Model (AWS SAM), AWS Cloud Development Kit (AWS CDK), or Terraform streamlines the process of defining serverless applications. AWS SAM, natively supported by CloudFormation, offers a simplified syntax for specifying serverless resources. AWS Lambda Layers help manage shared libraries across multiple Lambda functions, minimizing function footprint, centralizing tenant-aware libraries, and improving the developer experience. Lambda SnapStart for Java enhances startup performance for latency-sensitive applications.

To deploy, specify resources and permissions policies in a CloudFormation template, package deployment artifacts, and deploy the template. SAM Local, an AWS CLI tool, allows local development, testing, and analysis of serverless applications before uploading to Lambda.

Integration with tools like AWS Cloud9 IDE, AWS CodeBuild, AWS CodeDeploy, and AWS CodePipeline streamlines authoring, testing, debugging, and deploying SAM-based applications.

The following diagram shows deploying AWS Serverless Application Model resources using CloudFormation and AWS CI/CD tools.

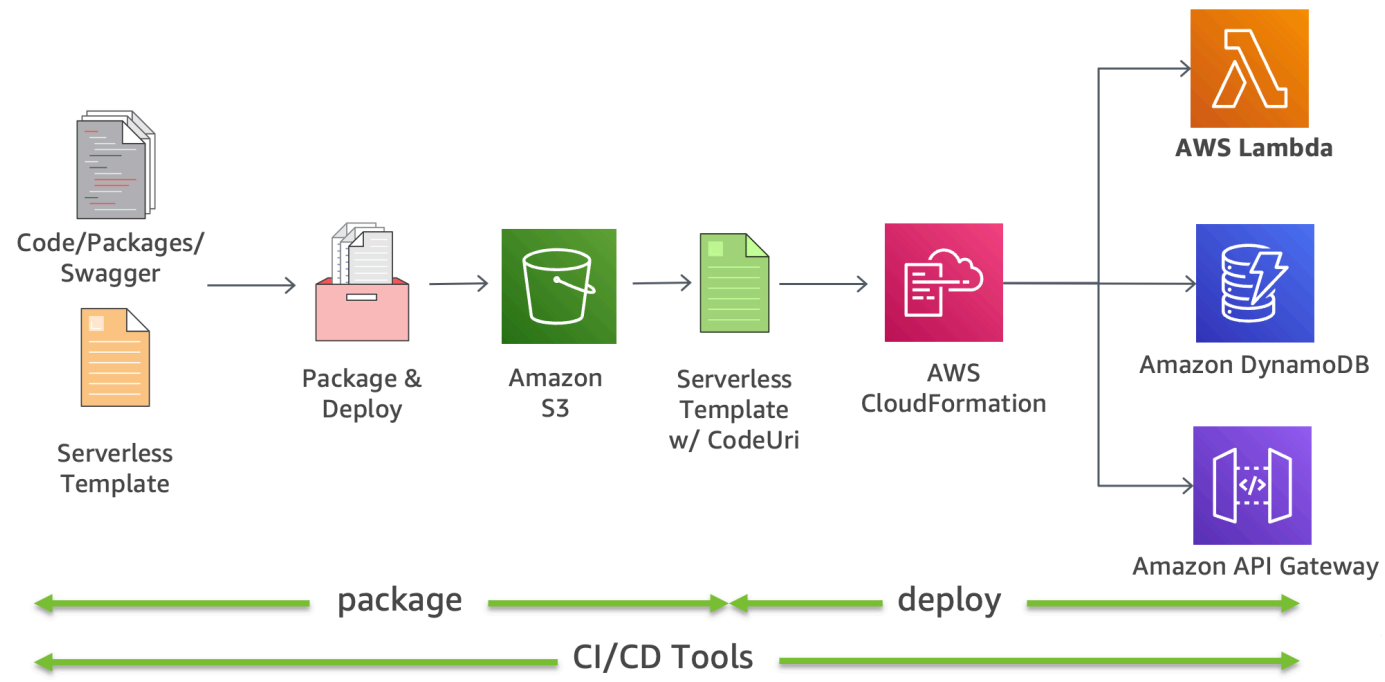

Figure 2: AWS Serverless Application Model (AWS SAM)

Abstracting multi-tenancy complexities

In a multi-tenant environment like SaaS platforms, it's crucial to streamline the intricacies related to multi-tenancy, freeing up developers to concentrate on feature and functionality development. This can be achieved using tools such as AWS Lambda Layers, which offer shared libraries for addressing cross-cutting concerns. The rationale behind this approach is that shared libraries and tools, when used correctly, efficiently manage tenant context.

However, they should not extend to encapsulating business logic due to the complexity and risk they may introduce. A fundamental issue with shared libraries is the increased complexity surrounding updates, making them more challenging to manage compared to standard code duplication. Thus, it's essential to strike a balance between the use of shared libraries and duplication in the quest for the most effective abstraction.

API management

Managing APIs can be time-consuming, especially when considering multiple versions, stages of the development cycle, authorization, and other features like throttling and caching. Apart from

API Gateway, some customers also use ALB (Application Load Balancer) or NLB (Network Load Balancer) for API management. Amazon API Gateway helps reduce the operational complexity of creating and maintaining RESTful APIs. It allows you to create APIs programmatically, serves as a "front door" to access data, business logic, or functionality from your backend services, Authorization and access control, rate limiting, caching, monitoring, and traffic management and runs APIs without managing servers.

Figure 3 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 reduce latency and provide an optimal user experience.

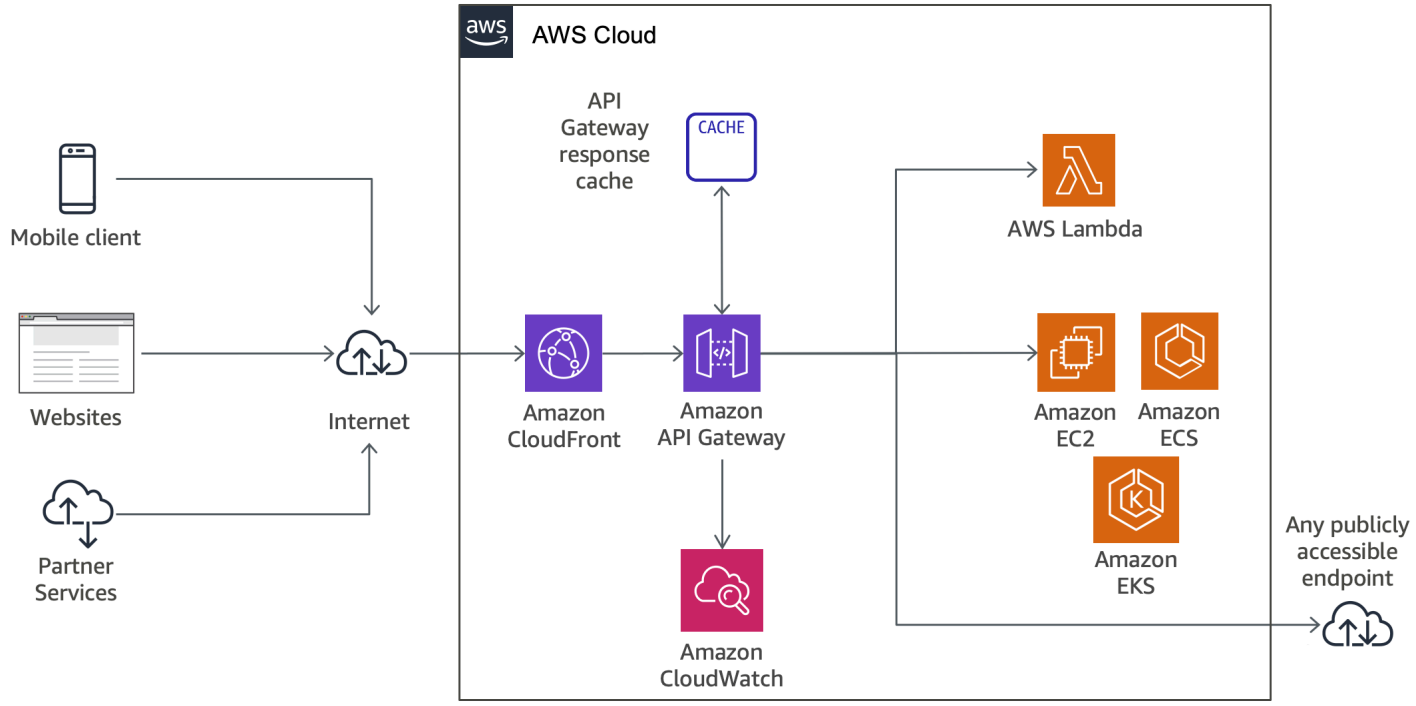

Figure 3: API Gateway call flow

Microservices on serverless technologies

Using microservices with serverless technologies can greatly decrease operational complexity. AWS Lambda and AWS Fargate, integrated with API Gateway, allows for the creation of fully serverless applications. As of April 7, 2023, Lambda functions can progressively stream response payloads back to the client, enhancing performance for web and mobile applications. Prior to this, Lambda-based applications using the traditional request-response invocation model had to fully generate and buffer the response before returning it to the client, which could delay the time to first byte. With response streaming, functions can send partial responses back to the client as they become ready, significantly improving the time to first byte, which web and mobile applications are especially sensitive to.

Figure 4 demonstrates a serverless microservice architecture using AWS Lambda and managed services. This serverless architecture mitigates the need to design for scale and high availability, and reduces the effort needed for running and monitoring the underlying infrastructure.

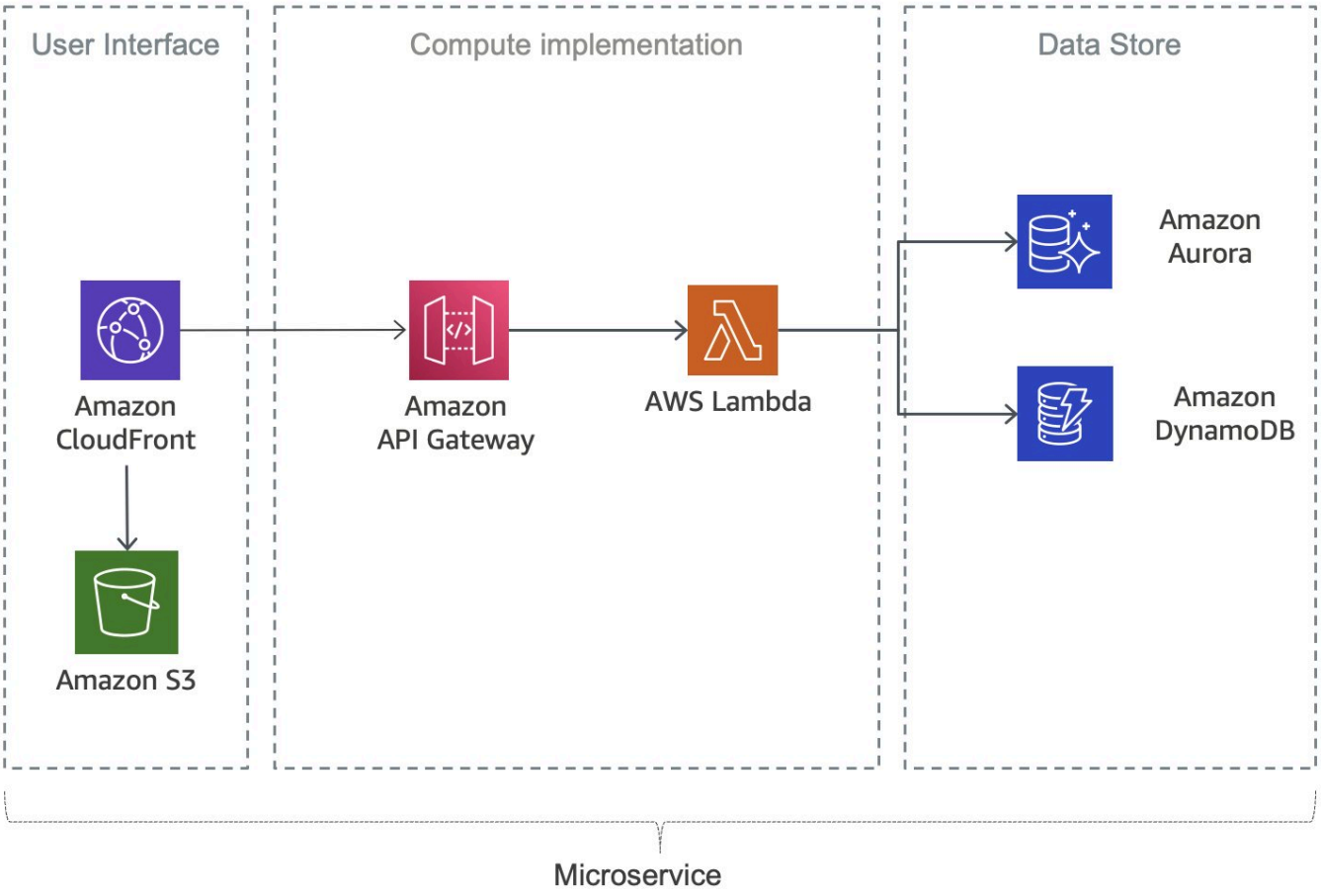

Figure 4: Serverless microservice using AWS Lambda

Figure 5 displays a similar serverless implementation using containers with AWS Fargate, removing concerns about underlying infrastructure. It also features Amazon Aurora Serverless, an on-demand, auto-scaling database that automatically adjusts capacity based on your application's requirements.

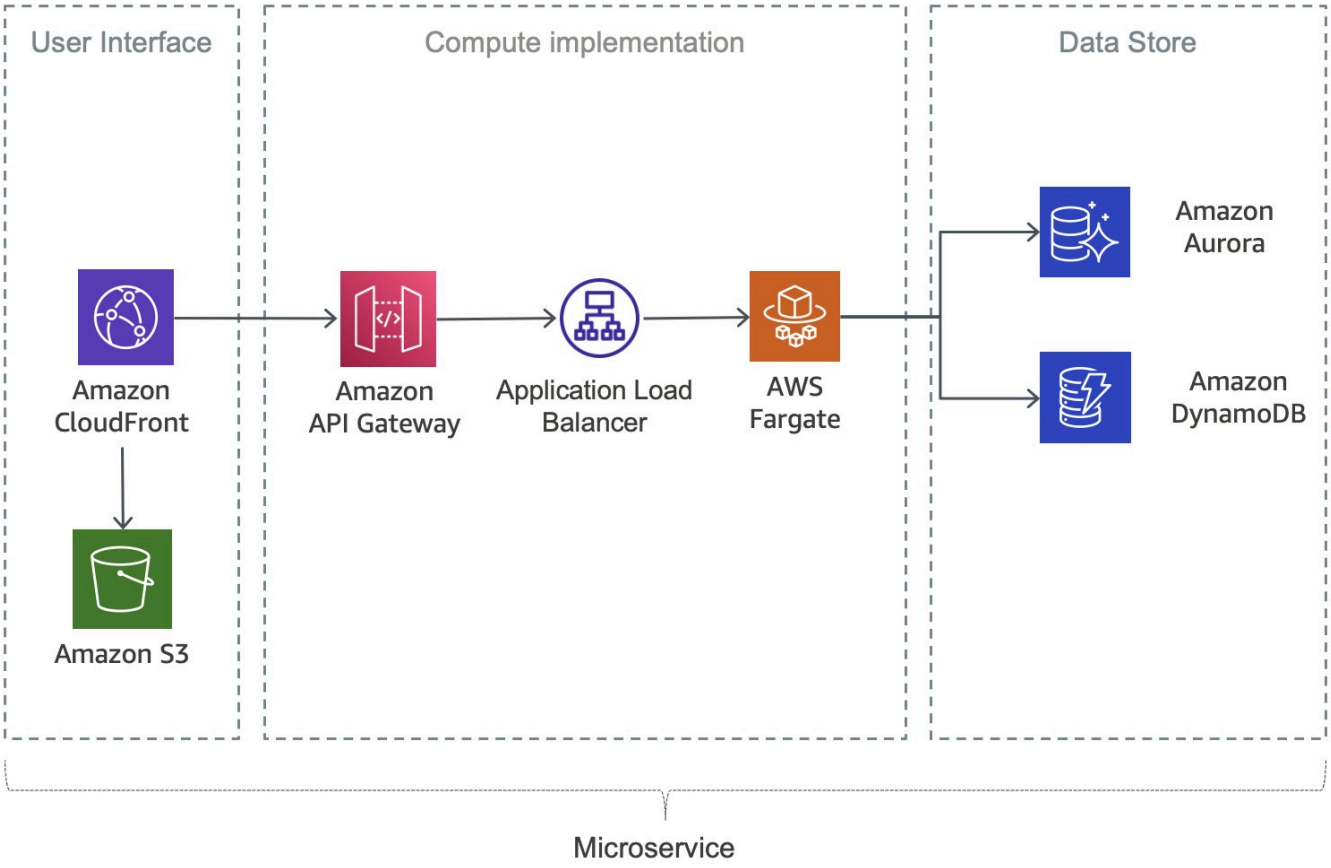

Figure 5: Serverless microservice using AWS Fargate

Resilient, efficient, and cost-optimized systems

Disaster recovery (DR)

Microservices applications often follow the Twelve-Factor Application patterns, where processes are stateless, and persistent data is stored in stateful backing services like databases. This simplifies disaster recovery (DR) because if a service fails, it's easy to launch new instances to restore functionality.

Disaster recovery strategies for microservices should focus on downstream services that maintain the application's state, such as file systems, databases, or queues. Organizations should plan for recovery time objective (RTO) and recovery point objective (RPO). RTO is the maximum acceptable delay between service interruption and restoration, while RPO is the maximum time since the last data recovery point.

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

High availability (HA)

We'll examine high availability (HA) for various components of a microservices architecture.

Amazon EKS ensures high availability by running Kubernetes control and data plane instances across multiple Availability Zones. It automatically detects and replaces unhealthy control plane instances and provides automated version upgrades and patching.

Amazon ECR uses Amazon Simple Storage Service (Amazon S3) for storage to make your container images highly available and accessible. It 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 a Region, offering multiple scheduling strategies that place containers for resource needs and availability requirements.

AWS Lambda operates in multiple Availability Zones, ensuring availability during service interruptions in a single zone. If connecting your function to a VPC, specify subnets in multiple Availability Zones for high availability.

Distributed systems components

In a microservices architecture, service discovery refers to the process of dynamically locating and identifying the network locations (IP addresses and ports) of individual microservices within a distributed system.

When choosing an approach on AWS, consider factors such as:

- **Code modification:** Can you get the benefits without modifying code?
- **Cross-VPC or cross-account traffic:** If required, does your system need efficient management of communication across different VPCs or AWS accounts?
- **Deployment strategies:** Does your system use or plan to use advanced deployment strategies such as blue-green or canary deployments?
- **Performance considerations:** If your architecture frequently communicates with external services, what will be the impact on overall performance?

AWS offers several methods for implementing service discovery in your microservices architecture:

- **Amazon ECS Service Discovery:** Amazon ECS supports service discovery using its DNS-based method or by integrating with AWS Cloud Map (see ECS Service discovery). ECS Service Connect further improves connection management, which can be especially beneficial for larger applications with multiple interacting services.
- **Amazon Route 53:** Route 53 integrates with ECS and other AWS services, such as EKS, to facilitate service discovery. In an ECS context, Route 53 can use the ECS Service Discovery feature, which leverages the Auto Naming API to automatically register and deregister services.
- **AWS Cloud Map:** This option offers a dynamic API-based service discovery, which propagates changes across your services.

For more advanced communication needs, AWS provides two service mesh options:

- **Amazon VPC Lattice** is an application networking service that consistently connects, monitors, and secures communications between your services, helping to improve productivity so that your developers can focus on building features that matter to your business. You can define policies for network traffic management, access, and monitoring to connect compute services in a simplified and consistent way across instances, containers, and serverless applications.

- **AWS App Mesh**: Based on the open-source Envoy proxy, App Mesh caters to advanced needs with sophisticated routing, load balancing, and comprehensive reporting. Unlike Amazon VPC Lattice, App Mesh does support the TCP protocol.

In case you're already using third-party software, such as HashiCorp Consul, or Netflix Eureka for service discovery, you might prefer to continue using these as you migrate to AWS, enabling a smoother transition.

The choice between these options should align with your specific needs. For simpler requirements, DNS-based solutions like Amazon ECS or AWS Cloud Map might be sufficient. For more complex or larger systems, service meshes like Amazon VPC Lattice or AWS App Mesh might be more suitable.

In conclusion, designing a microservices architecture on AWS is all about selecting the right tools to meet your specific needs. By keeping in mind the considerations discussed, you can ensure you're making informed decisions to optimize your system's service discovery and inter-service communication.

Distributed data management

In traditional applications, all components often share a single database. In contrast, each component of a microservices-based application maintains its own data, promoting independence and decentralization. This approach, known as distributed data management, brings new challenges.

One such challenge arises from the trade-off between consistency and performance in distributed systems. It's often more practical to accept slight delays in data updates (eventual consistency) than to insist on instant updates (immediate consistency).

Sometimes, business operations require multiple microservices to work together. If one part fails, you might have to undo some completed tasks. The Saga pattern helps manage this by coordinating a series of compensating actions.

To help microservices stay in sync, a centralized data store can be used. This store, managed with tools like AWS Lambda, AWS Step Functions, and Amazon EventBridge, can assist in cleaning up and deduplicating data.

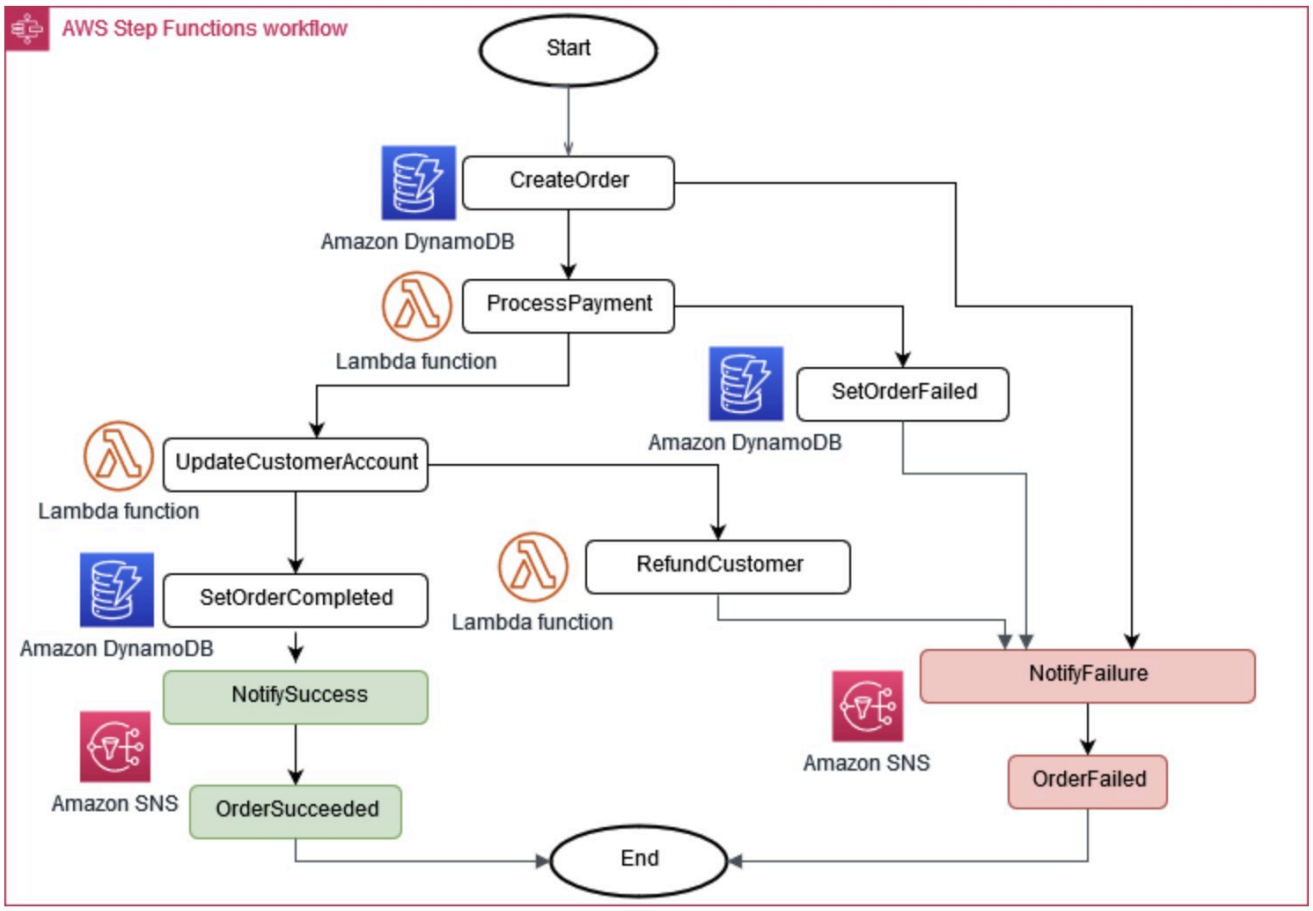

Figure 6: Saga execution coordinator

A common approach in managing changes across microservices is *event sourcing*. Every change in the application is recorded as an event, creating a timeline of the system's state. This approach not only helps debug and audit but also allows different parts of an application to react to the same events.

Event sourcing often works hand-in-hand with the Command Query Responsibility Segregation (CQRS) pattern, which separates data modification and data querying into different modules for better performance and security.

On AWS, you can implement these patterns using a combination of services. As you can see in Figure 7, Amazon Kinesis Data Streams can serve as your central event store, while Amazon S3 provides a durable storage for all event records. AWS Lambda, Amazon DynamoDB, and Amazon API Gateway work together to handle and process these events.

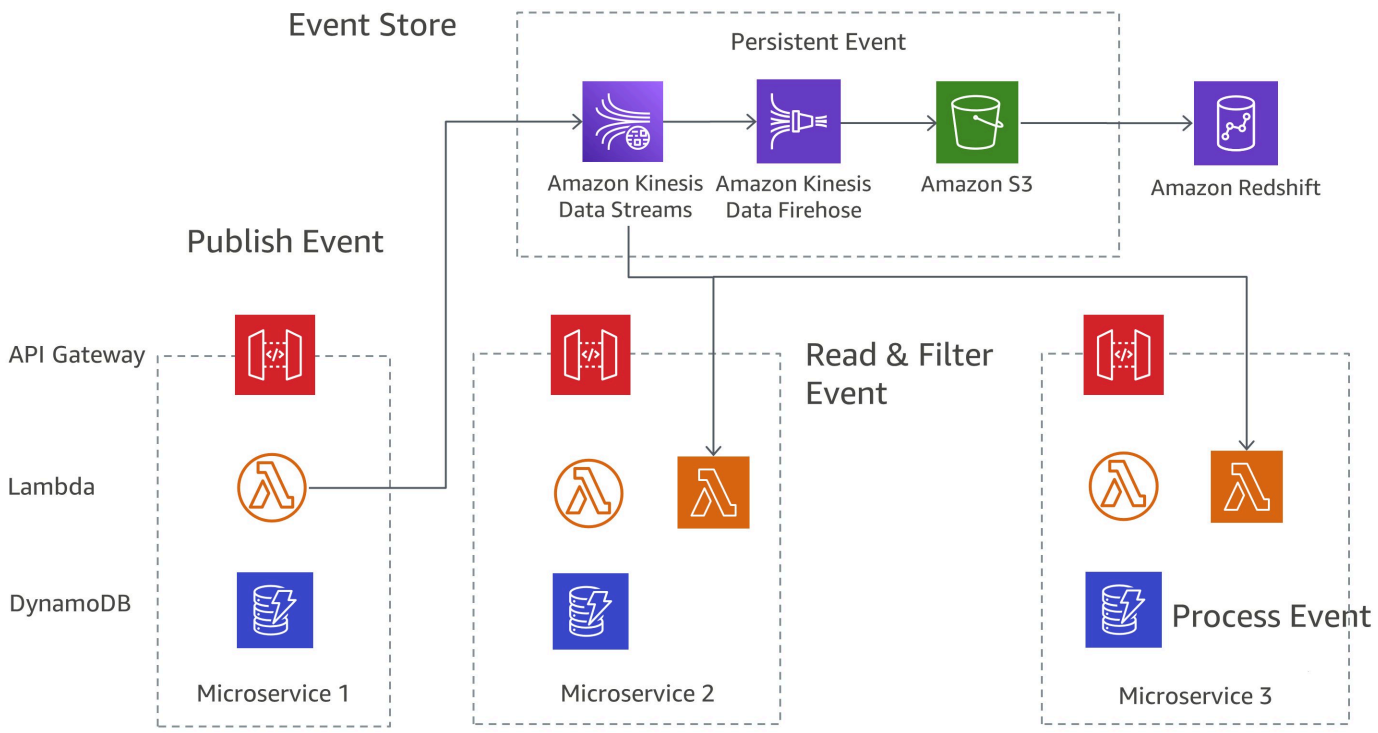

Figure 7: Event sourcing pattern on AWS

Remember, in distributed systems, events might be delivered multiple times due to retries, so it's important to design your applications to handle this.

Configuration management

In a microservices architecture, each service interacts with various resources like databases, queues, and other services. A consistent way to configure each service's connections and operating environment is vital. Ideally, an application should adapt to new configurations without needing a restart. This approach is part of the Twelve-Factor App principles, which recommend storing configurations in environment variables.

A different approach is to use AWS App Config. It's a feature of AWS Systems Manager which makes it easy for customers to quickly and safely configure, validate, and deploy feature flags and application configuration. Your feature flag and configurations data can be validated syntactically or semantically in the pre-deployment phase, and can be monitored and automatically rolled back if an alarm that you have configured is triggered. AppConfig can be integrated with Amazon ECS and Amazon EKS by using the AWS AppConfig agent. The agent functions as a sidecar container running alongside your Amazon ECS and Amazon EKS container applications. If you use AWS AppConfig feature flags or other dynamic configuration data in a Lambda function, then we recommend that you add the AWS AppConfig Lambda extension as a layer to your Lambda function.

GitOps is an innovative approach to configuration management that uses Git as the source of truth for all configuration changes. This means that any changes made to your configuration files are automatically tracked, versioned, and audited through Git.

Secrets management

Security is paramount, so credentials should not be passed in plain text. AWS offers secure services for this, like AWS Systems Manager Parameter Store and AWS Secrets Manager. These tools can send secrets to containers in Amazon EKS as volumes, or to Amazon ECS as environment variables. In AWS Lambda, environment variables are made available to your code automatically. For Kubernetes workflows, the External Secrets Operator fetches secrets directly from services like AWS Secrets Manager, creating corresponding Kubernetes Secrets. This enables a seamless integration with Kubernetes-native configurations.

Cost optimization and sustainability

Microservices architecture can enhance cost optimization and sustainability. By breaking an application into smaller parts, you can scale up only the services that need more resources, reducing cost and waste. This is particularly useful when dealing with variable traffic. Microservices are independently developed. So developers can do smaller updates, and reduce the resources spent on end to end testing. While updating they will have to test only a subset of the features as opposed to monoliths.

Stateless components (services that store state in an external data store instead of a local data store) in your architecture can make use of Amazon EC2 Spot Instances, which offer unused EC2 capacity in the AWS cloud. These instances are more cost efficient than on-demand instances and are perfect for workloads that can handle interruptions. This can further cut costs while maintaining high availability.

With isolated services, you can use cost-optimized compute options for each auto-scaling group. For example, AWS Graviton offers cost-effective, high-performance compute options for workloads that suit ARM-based instances.

Optimizing costs and resource usage also helps minimize environmental impact, aligning with the Sustainability pillar of the Well-Architected Framework. You can monitor your progress in reducing carbon emissions using the AWS Customer Carbon Footprint Tool. This tool provides insights into the environmental impact of your AWS usage.

Communication mechanisms

In the microservices paradigm, various components of an application need to communicate over a network. Common approaches for this include REST-based, GraphQL-based, gRPC-based and asynchronous messaging.

Topics

REST-based communication

The HTTP/S protocol, used broadly for synchronous communication between microservices, often operates through RESTful APIs. AWS's API Gateway offers a streamlined way to build an API that serves as a centralized access point to backend services, handling tasks like traffic management, authorization, monitoring, and version control.

GraphQL-based communication

Similarly, GraphQL is a widespread method for synchronous communication, using the same protocols as REST but limiting exposure to a single endpoint. With AWS AppSync, you can create and publish GraphQL applications that interact with AWS services and datastores directly, or incorporate Lambda functions for business logic.

gRPC-based communication

gRPC is a synchronous, lightweight, high performance, open-source RPC communication protocol. gRPC improves upon its underlying protocols by using HTTP/2 and enabling more features such as compression and stream prioritization. It uses Protobuf Interface Definition Language (IDL) which is binary-encoded and thus takes advantage of HTTP/2 binary framing.

Asynchronous messaging and event passing

Asynchronous messaging allows services to communicate by sending and receiving messages via a queue. This enables services to remain loosely coupled and promote service discovery.

Messaging can be defined of the following three types:

- **Message Queues:** A message queue acts as a buffer that decouples senders (producers) and receivers (consumers) of messages. Producers enqueue messages into the queue, and consumers dequeue and process them. This pattern is useful for asynchronous communication, load leveling, and handling bursts of traffic.
- **Publish-Subscribe:** In the publish-subscribe pattern, a message is published to a topic, and multiple interested subscribers receive the message. This pattern enables broadcasting events or messages to multiple consumers asynchronously.
- **Event-Driven Messaging:** Event-driven messaging involves capturing and reacting to events that occur in the system. Events are published to a message broker, and interested services subscribe to specific event types. This pattern enables loose coupling and allows services to react to events without direct dependencies.

To implement each of these message types, AWS offers various managed services such as Amazon SQS, Amazon SNS, Amazon EventBridge, Amazon MQ, and Amazon MSK. These services have unique features tailored to specific needs:

- **Amazon Simple Queue Service (Amazon SQS) and Amazon Simple Notification Service (Amazon SNS):** As you can see in Figure 8, these two services complement each other, with Amazon SQS providing a space for storing messages and Amazon SNS enabling delivery of messages to multiple subscribers. They are effective when the same message needs to be delivered to multiple destinations.

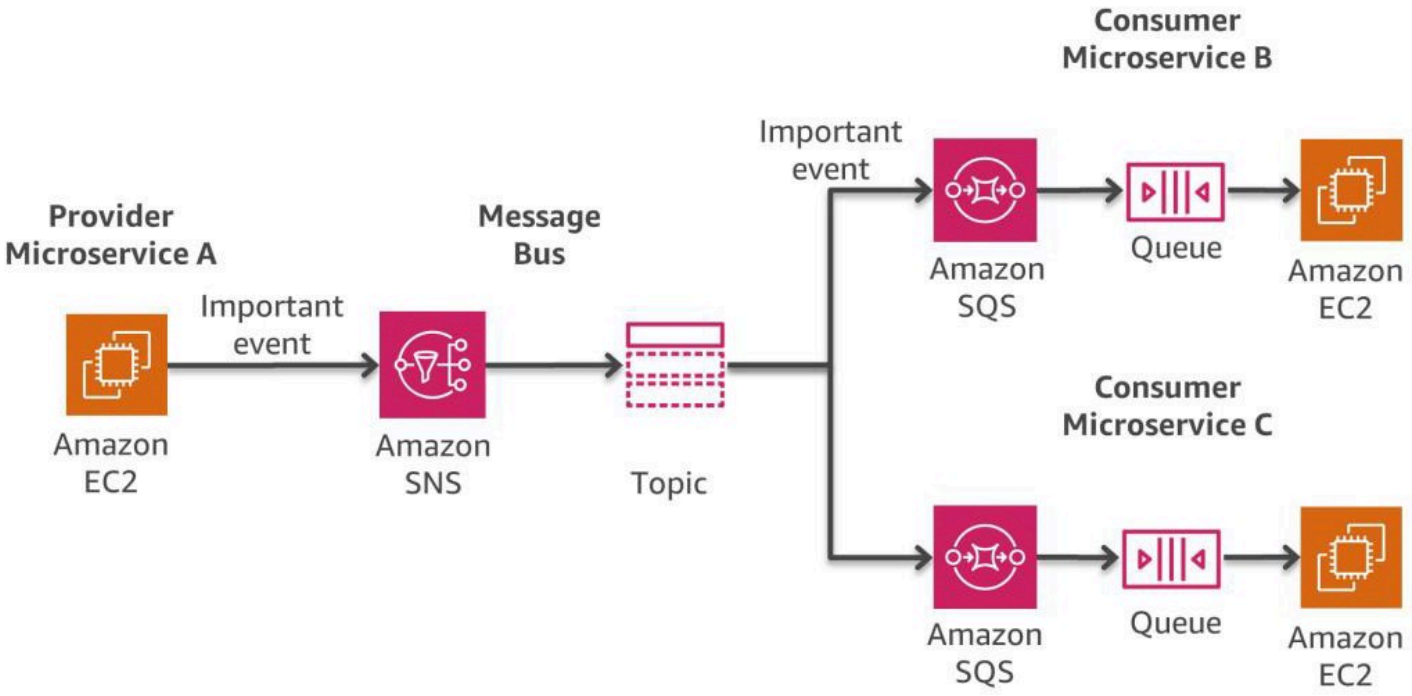

Figure 8: Message bus pattern on AWS

- **Amazon EventBridge:** a serverless service that uses events to connect application components together, making it easier for you to build scalable event-driven applications. Use it to route events from sources such as home-grown applications, AWS services, and third- party software to consumer applications across your organization. EventBridge provides a simple and consistent way to ingest, filter, transform, and deliver events so you can build new applications quickly. EventBridge event buses are well suited for many-to-many routing of events between event-driven services.
- **Amazon MQ:** a good choice if you have a pre-existing messaging system that uses standard protocols like JMS, AMQP, etc. This managed service provides a replacement for your system without disrupting operations.
- **Amazon MSK (Managed Kafka):** a messaging system for storing and reading messages, useful for cases where messages need to be processed multiple times. It also supports real-time message streaming.
- **Amazon Kinesis:** real-time processing and analyzing of streaming data. This allows for the development of real-time applications and provides seamless integration with the AWS ecosystem.

Remember, the best service for you depends on your specific needs, so it's important to understand what each one offers and how they align with your requirements.

Orchestration and state management

Microservices orchestration refers to a centralized approach, where a central component, known as the orchestrator, is responsible for managing and coordinating the interactions between microservices. Orchestrating workflows across multiple microservices can be challenging. Embedding orchestration code directly into services is discouraged, as it introduces tighter coupling and hinders replacing individual services.

Step Functions provides a workflow engine to manage service orchestration complexities, such as error handling and serialization. This allows you to scale and change applications quickly without adding coordination code. Step Functions is part of the AWS serverless platform and supports Lambda functions, Amazon EC2, Amazon EKS, Amazon ECS, SageMaker, AWS Glue, and more.

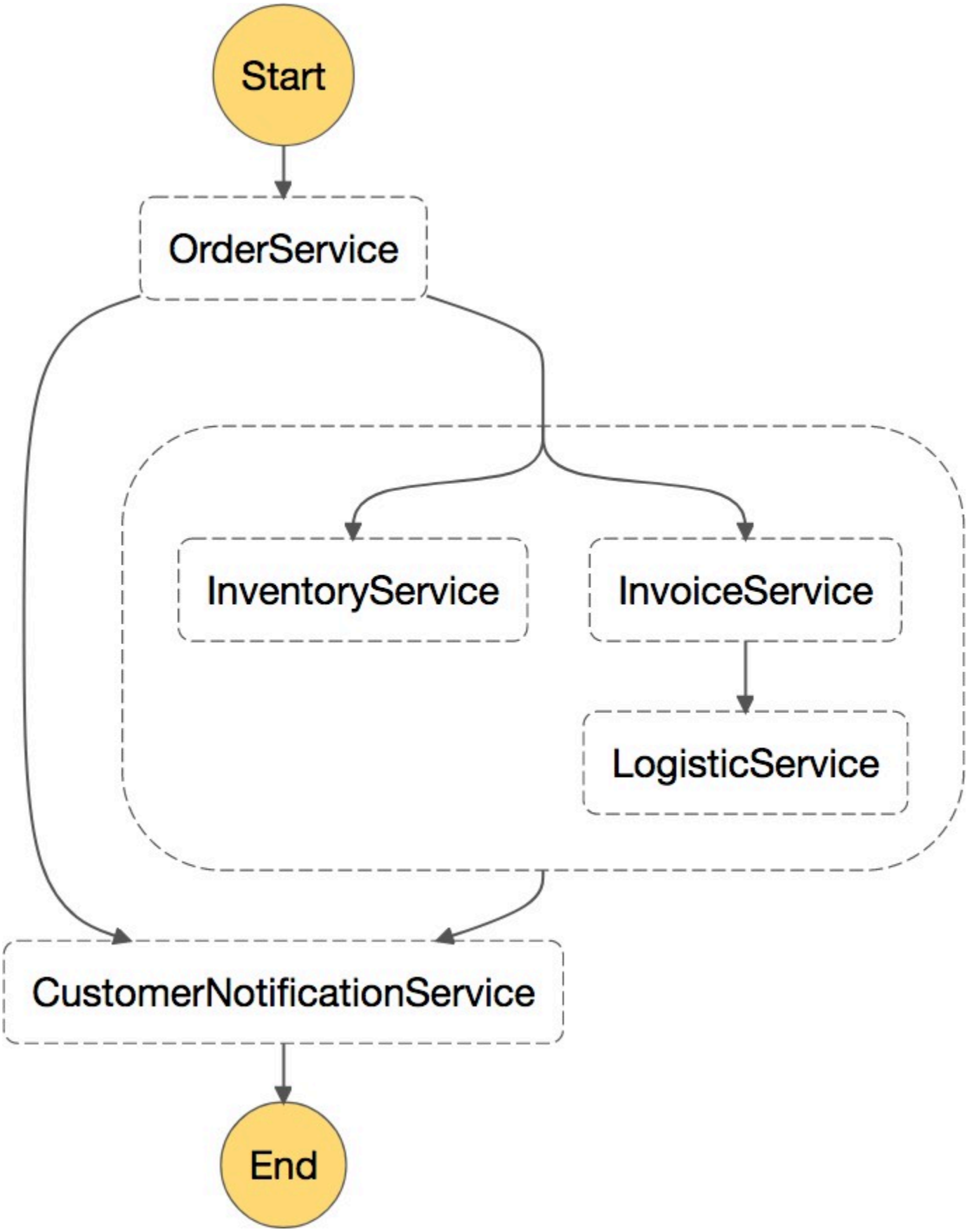
Start
OrderService
InventoryService
InvoiceService
LogisticService
CustomerNotificationService
End

Figure 9: An example of a microservices workflow with parallel and sequential steps invoked by AWS Step Functions

Amazon Managed Workflows for Apache Airflow (Amazon MWAA) is an alternative to Step Functions. You should use Amazon MWAA if you prioritize open source and portability. Airflow has a large and active open-source community that contributes new functionality and integrations regularly.

Observability

Since microservices architectures are inherently made up of many distributed components, observability across all those components becomes critical. Amazon CloudWatch enables this, collecting and tracking metrics, monitoring log files, and reacting to changes in your AWS environment. It can monitor AWS resources and custom metrics generated by your applications and services.

Topics

Monitoring

CloudWatch offers system-wide visibility into resource utilization, application performance, and operational health. In a microservices architecture, custom metrics monitoring via CloudWatch is beneficial, as developers can choose which metrics to collect. Dynamic scaling can also be based on these custom metrics.

CloudWatch Container Insights extends this functionality, automatically collecting metrics for many resources like CPU, memory, disk, and network. It helps in diagnosing container-related issues, streamlining resolution.

For Amazon EKS, an often-preferred choice is Prometheus, an open-source platform providing comprehensive monitoring and alerting capabilities. It's typically coupled with Grafana for intuitive metrics visualization. Amazon Managed Service for Prometheus (AMP) offers a monitoring service fully compatible with Prometheus, letting you oversee containerized applications effortlessly. Additionally, Amazon Managed Grafana (AMG) simplifies the analysis and visualization of your metrics, eliminating the need for managing underlying infrastructure.

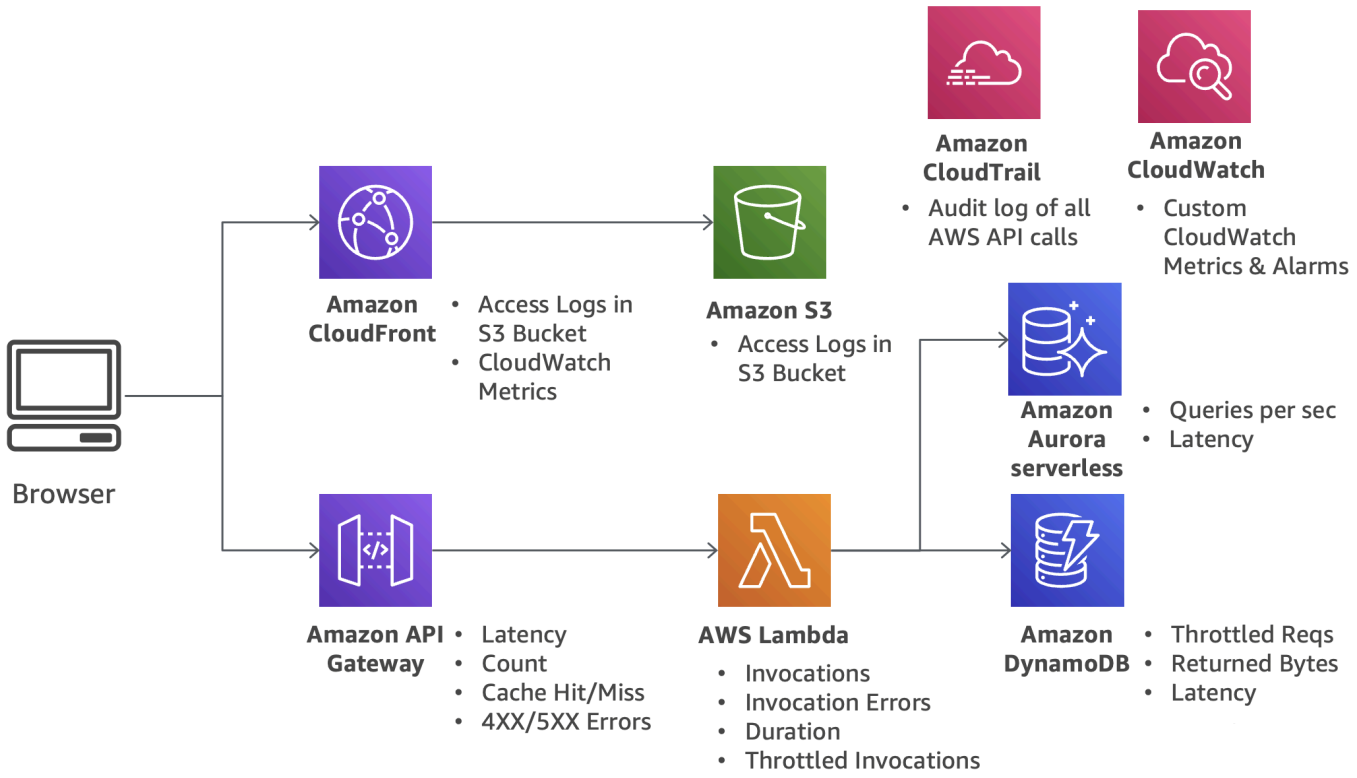

Figure 10: A serverless architecture with monitoring components

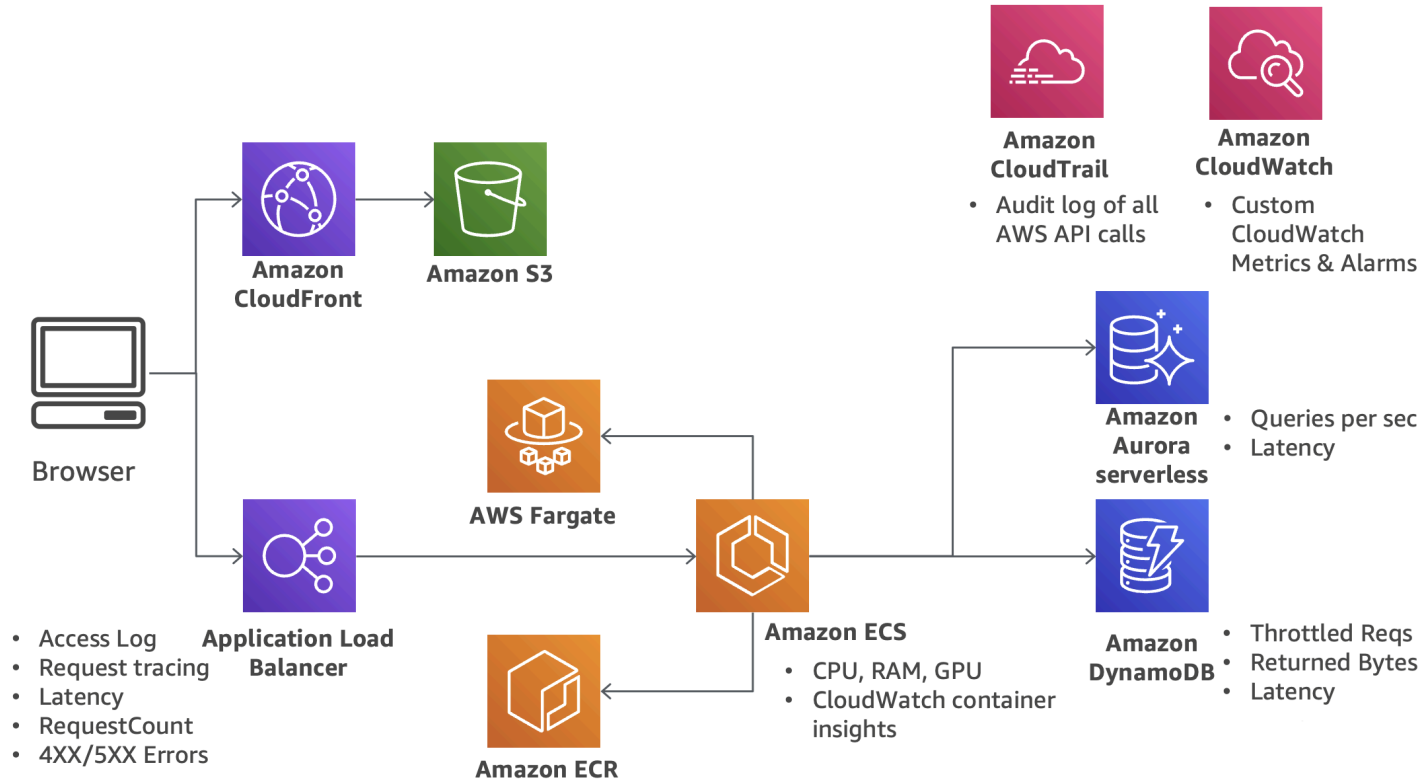

Figure 11: A container-based architecture with monitoring components

Centralizing logs

Logging is key to pinpoint and resolve issues. With microservices, you can release more frequently and experiment with new features. AWS provides services like Amazon S3, CloudWatch Logs, and Amazon OpenSearch Service to centralize log files. Amazon EC2 uses a daemon for sending logs to CloudWatch, while Lambda and Amazon ECS natively send their log output there. For Amazon EKS, either Fluent Bit or Fluentd can be used to forward logs to CloudWatch for reporting using OpenSearch and Kibana. However, due to the smaller footprint and performance advantages, Fluent Bit is recommended over Fluentd.

Figure 12 illustrates how logs from various AWS services are directed to Amazon S3 and CloudWatch. These centralized logs can be further analyzed using Amazon OpenSearch Service, inclusive of Kibana for data visualization. Also, Amazon Athena can be employed for ad hoc queries against the logs stored in Amazon S3.

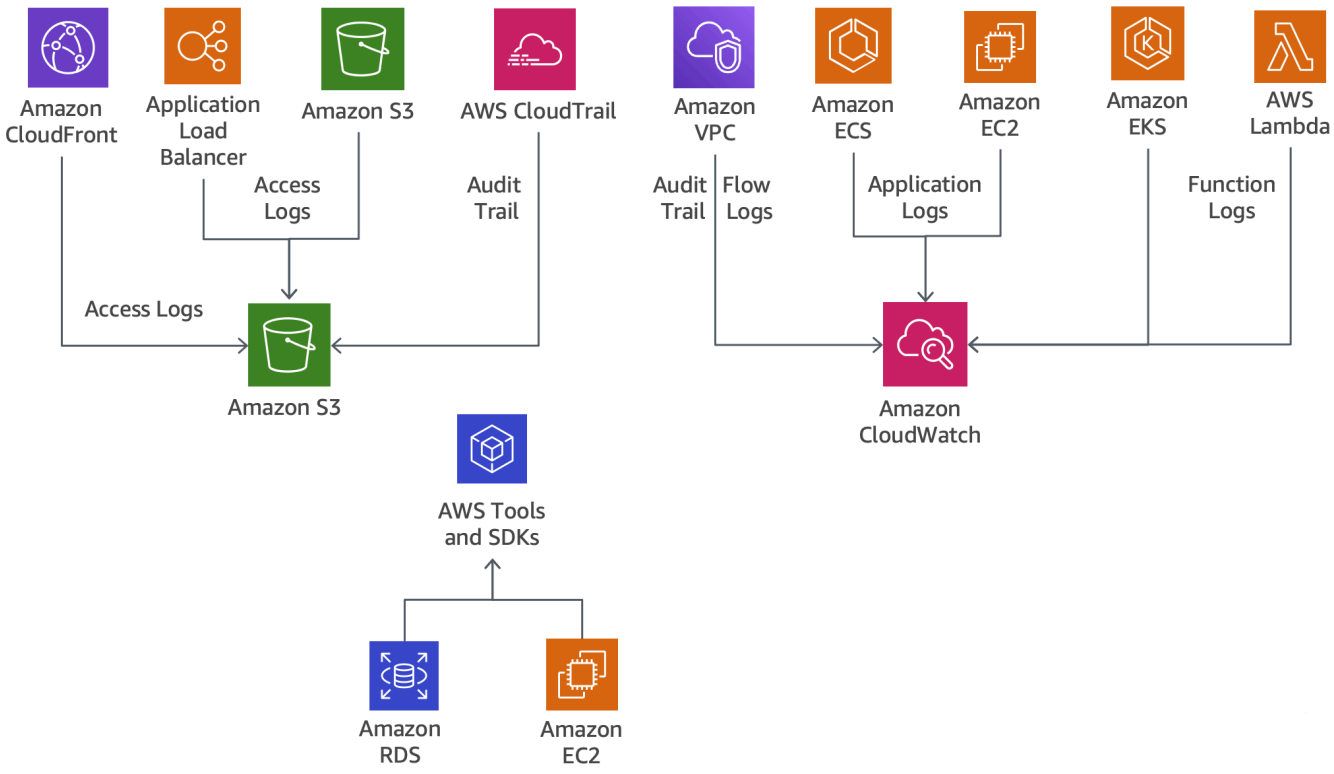

Figure 12: Logging capabilities of AWS services

Distributed tracing

Microservices often work together to handle requests. AWS X-Ray uses correlation IDs to track requests across these services. X-Ray works with Amazon EC2, Amazon ECS, Lambda, and Elastic Beanstalk.

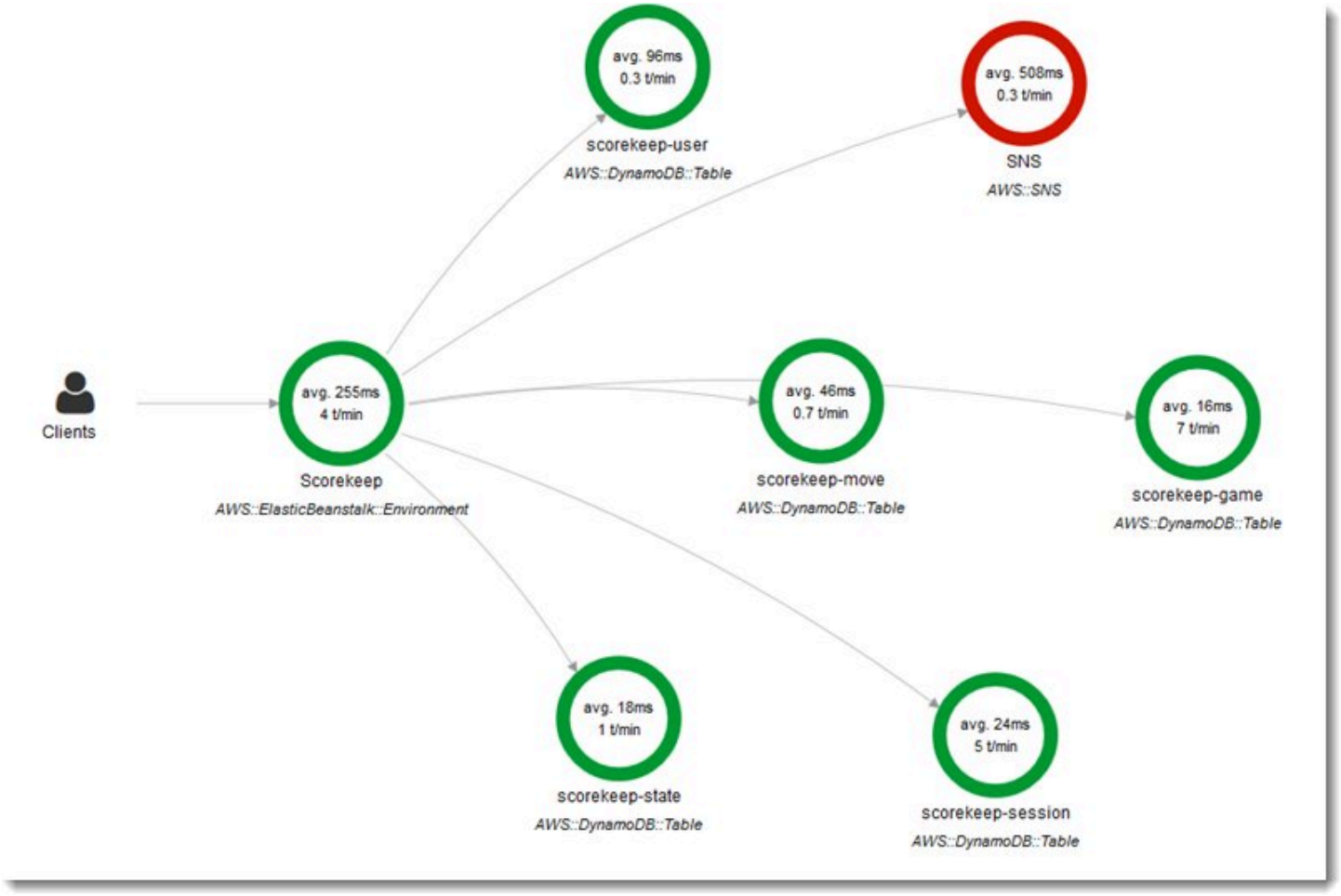

Figure 13: AWS X-Ray service map

AWS Distro for OpenTelemetry is part of the OpenTelemetry project and provides open-source APIs and agents to gather distributed traces and metrics, improving your application monitoring. It sends metrics and traces to multiple AWS and partner monitoring solutions. By collecting metadata from your AWS resources, it aligns application performance with the underlying infrastructure data, accelerating problem-solving. Plus, it's compatible with a variety of AWS services and can be used on-premises.

Log analysis on AWS

Amazon CloudWatch Logs Insights allows for real-time log exploration, analysis, and visualization. For further log file analysis, Amazon OpenSearch Service, which includes Kibana, is a powerful tool. CloudWatch Logs can stream log entries to OpenSearch Service in real time. Kibana, seamlessly integrated with OpenSearch, visualizes this data and offers an intuitive search interface.

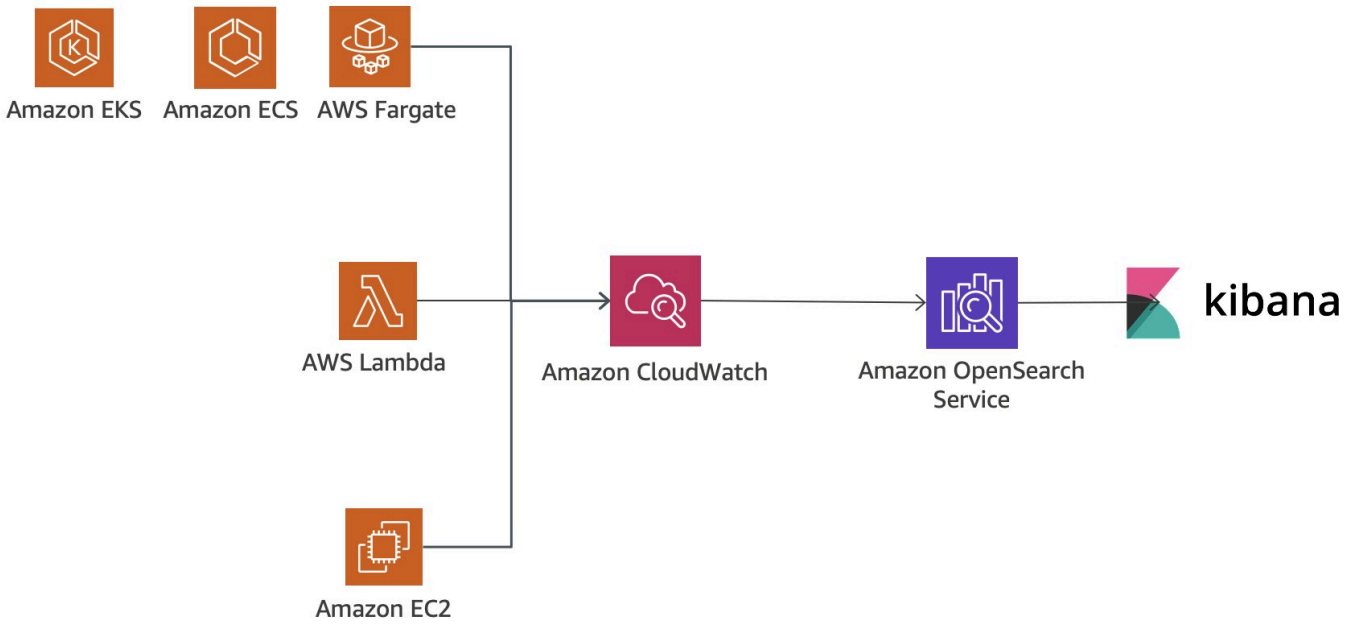

Figure 14: Log analysis with Amazon OpenSearch Service

Other options for analysis

For further log analysis, Amazon Redshift, a fully-managed data warehouse service, and Amazon QuickSight, a scalable business intelligence service, offer effective solutions. QuickSight provides easy connectivity to various AWS data services such as Redshift, RDS, Aurora, EMR, DynamoDB, Amazon S3, and Kinesis, simplifying data access.

CloudWatch Logs has the capability to stream log entries to Amazon Data Firehose, a service for delivering real-time streaming data. QuickSight then utilizes the data stored in Redshift for comprehensive analysis, reporting, and visualization.

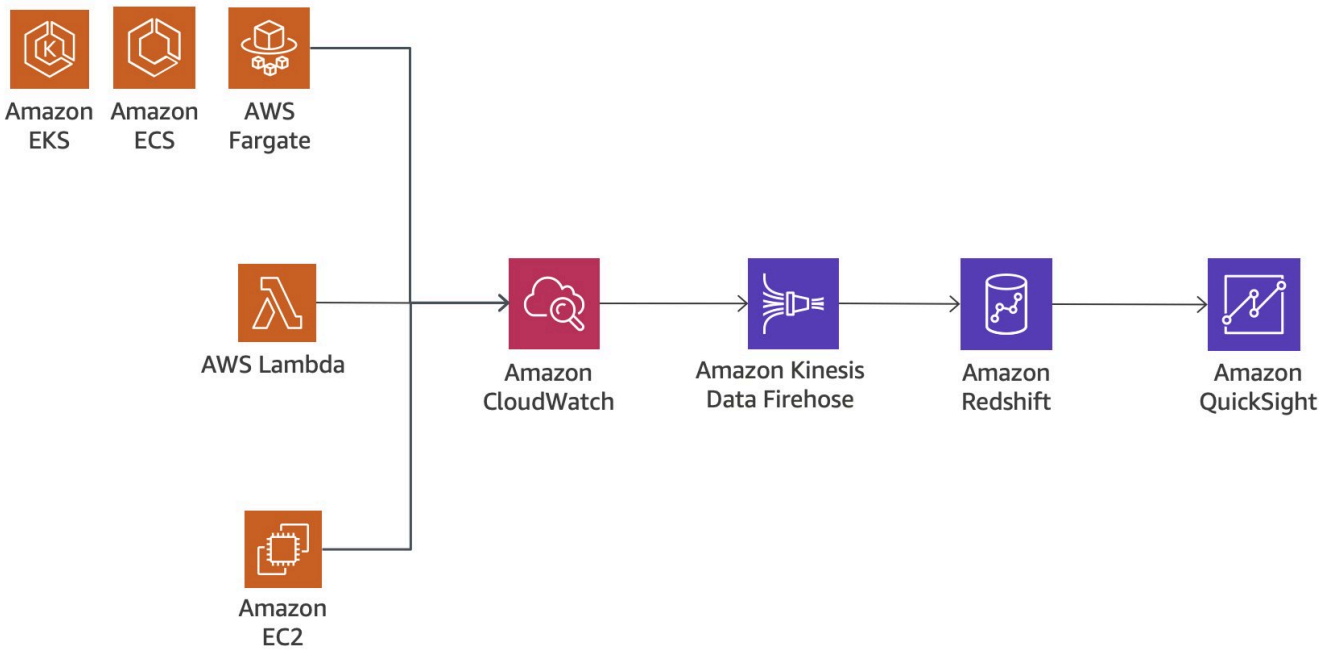

Figure 15: Log analysis with Amazon Redshift and Amazon QuickSight

Moreover, when logs are stored in S3 buckets, an object storage service, the data can be loaded into services like Redshift or EMR, a cloud-based big data platform, allowing for thorough analysis of the stored log data.

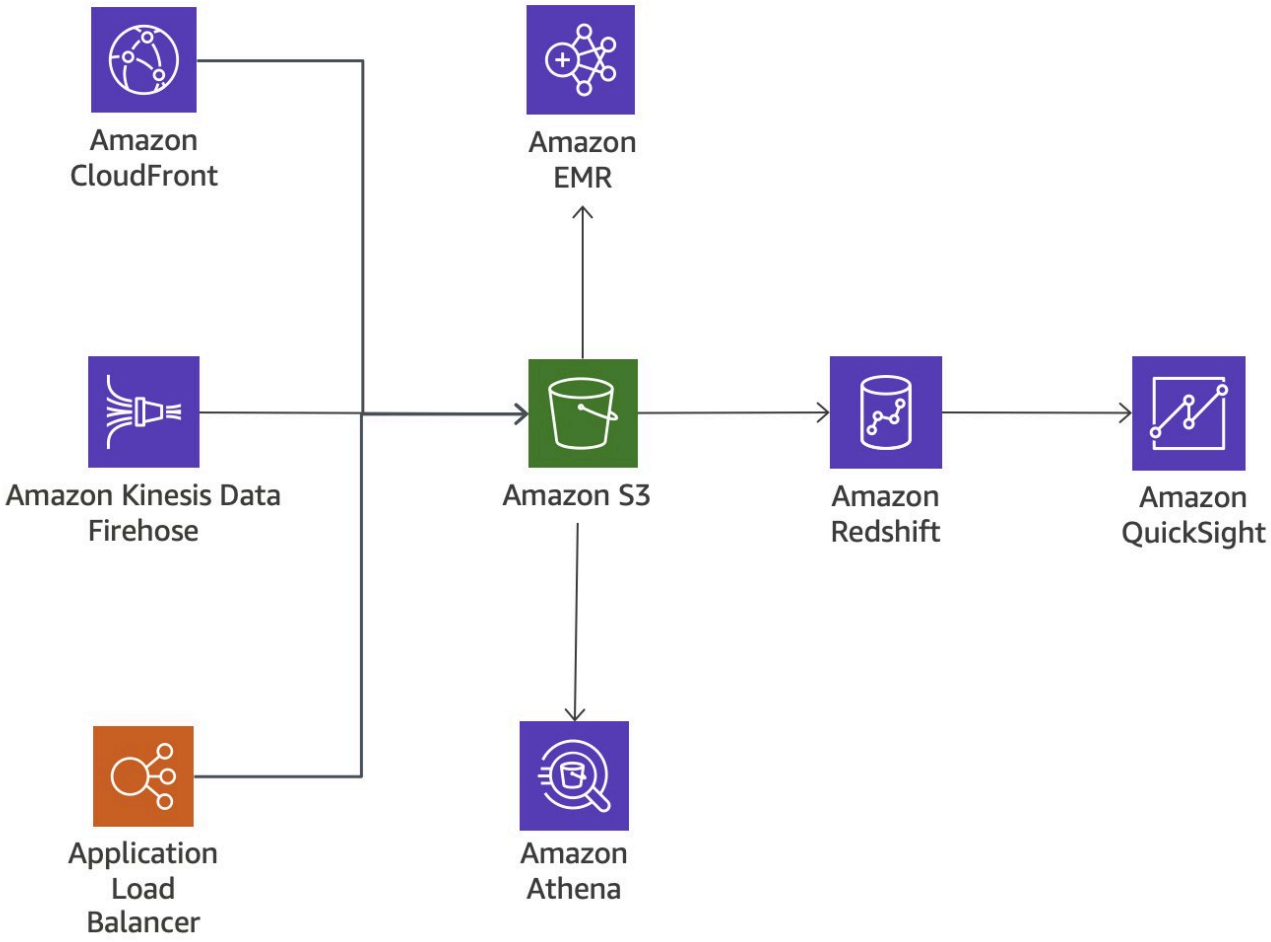

Figure 16: Streamlining Log Analysis: From AWS services to QuickSight

Managing chattiness in microservices communication

Chattiness refers to excessive communication between microservices, which can cause inefficiency due to increased network latency. It's essential to manage chattiness effectively for a well-functioning system.

Some key tools for managing chattiness are REST APIs, HTTP APIs and gRPC APIs. REST APIs offer a range of advanced features such as API keys, per-client throttling, request validation, AWS WAF integration, or private API endpoints. HTTP APIs are designed with minimal features and hence come at a lower price. For more details on this topic and a list of core features that are available in REST APIs and HTTP APIs, see Choosing between REST APIs and HTTP APIs.

Often, microservices use REST over HTTP for communication due to its widespread use. But in high-volume situations, REST's overhead can cause performance issues. It’s because the communication uses TCP handshake which is required for every new request. In such cases, gRPC API is a better choice. gRPC reduces the latency as it allows multiple requests over a single TCP connection. gRPC also supports bi-directional streaming, allowing clients and servers to send and receive messages at the same time. This leads to more efficient communication, especially for large or real-time data transfers.

If chattiness persists despite choosing the right API type, it may be necessary to reevaluate your microservices architecture. Consolidating services or revising your domain model could reduce chattiness and improve efficiency.

Using protocols and caching

Microservices often use protocols like gRPC and REST for communication (see the previous discussion on Communication mechanisms.) gRPC uses HTTP/2 for transport, while REST typically uses HTTP/1.1. gRPC employs protocol buffers for serialization, while REST usually uses JSON or XML. To reduce latency and communication overhead, caching can be applied. Services like Amazon ElastiCache or the caching layer in API Gateway can help reduce the number of calls between microservices.

Auditing

In a microservices architecture, it's crucial to have visibility into user actions across all services. AWS provides tools like AWS CloudTrail, which logs all API calls made in AWS, and AWS CloudWatch, which is used to capture application logs. This allows you to track changes and analyze behavior across your microservices. Amazon EventBridge can react to system changes quickly, notifying the right people or even automatically starting workflows to resolve issues.

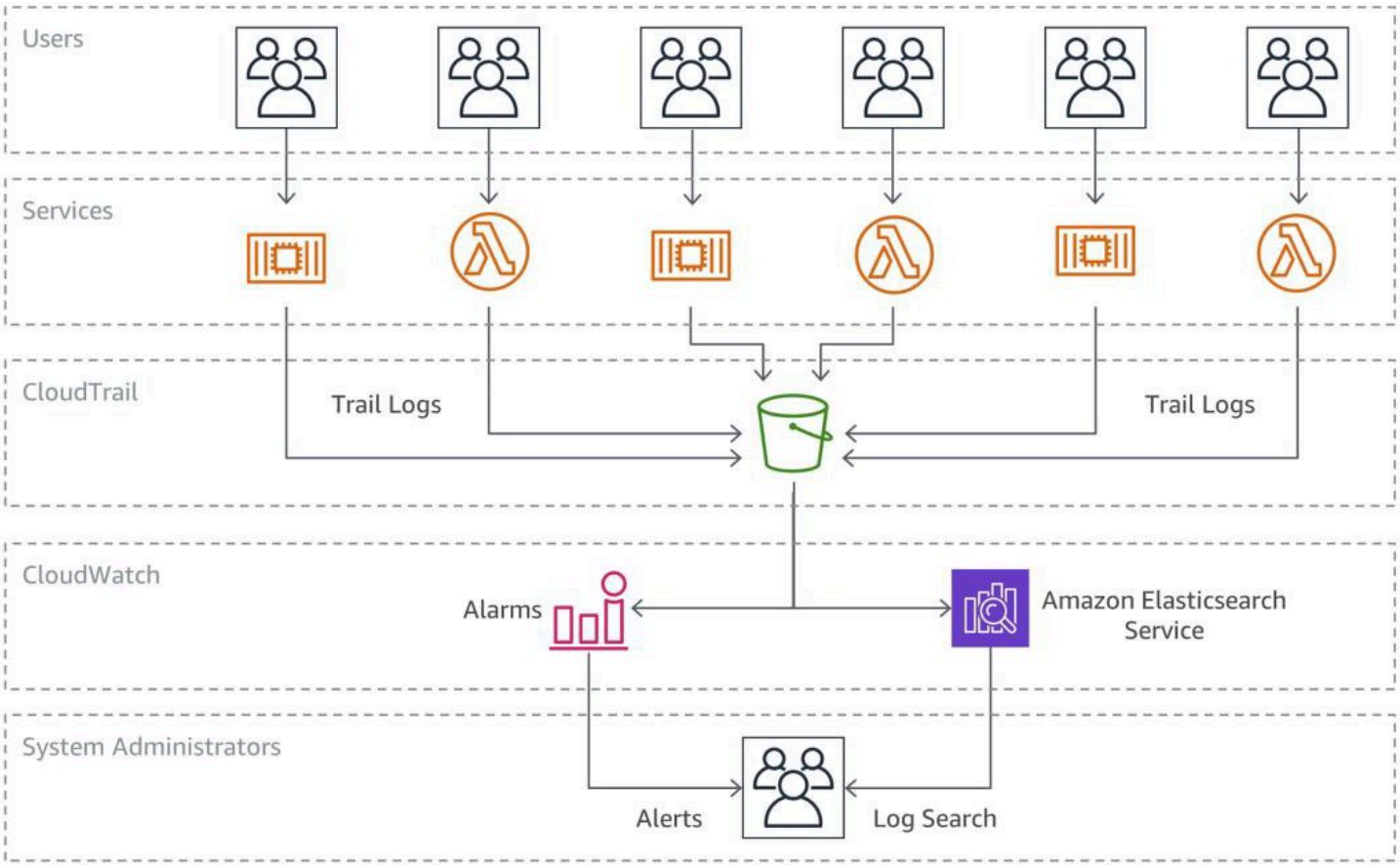

Figure 17: Auditing and remediation across your microservices

Resource inventory and change management

In an agile development environment with rapidly evolving infrastructure configurations, automated auditing and control are vital. AWS Config Rules provide a managed approach to monitoring these changes across microservices. They enable the definition of specific security policies that automatically detect, track, and send alerts on policy violations.

For instance, if an API Gateway configuration in a microservice is altered to accept inbound HTTP traffic instead of only HTTPS requests, a predefined AWS Config rule can detect this security violation. It logs the change for auditing and triggers an SNS notification, restoring the compliant state.

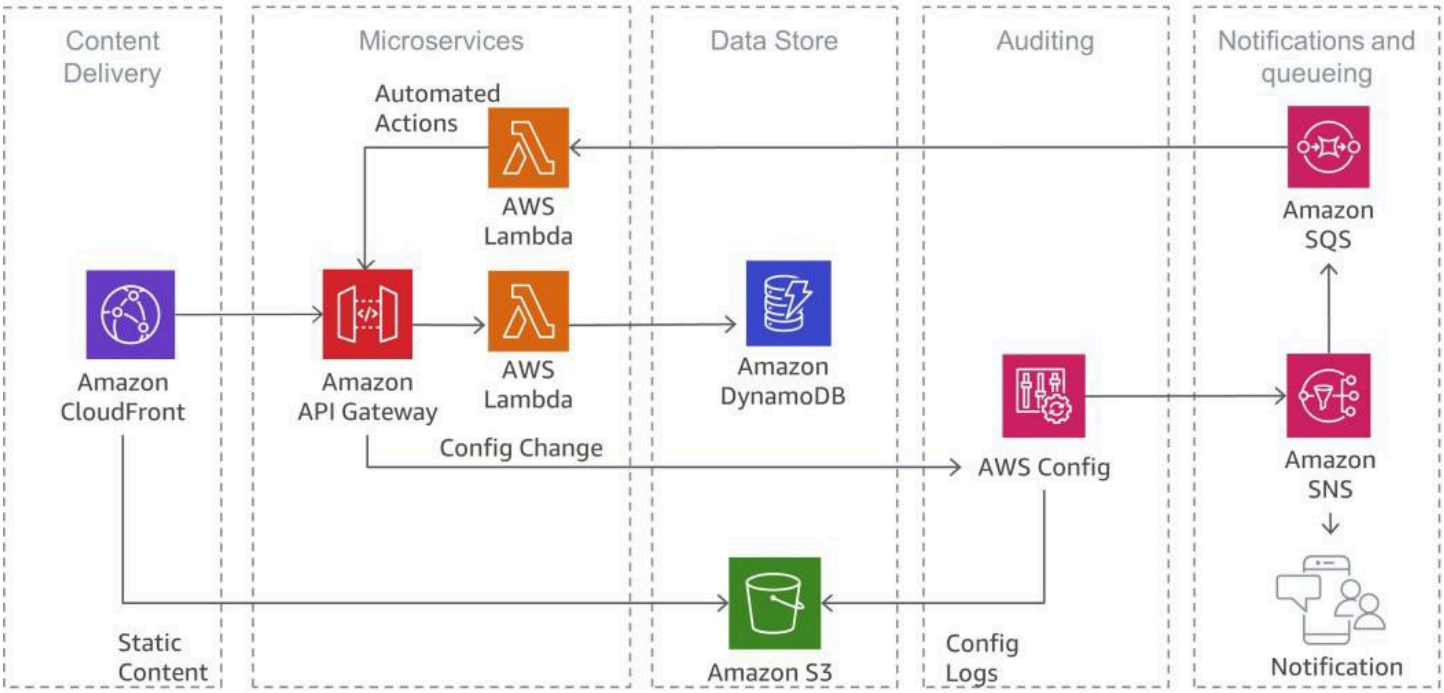

Figure 18: Detecting security violations with AWS Config

Conclusion

Microservices architecture, a versatile design approach that provides an alternative to traditional monolithic systems, assists in scaling applications, boosting development speed, and fostering organizational growth. With its adaptability, it can be implemented using containers, serverless approaches, or a blend of the two, tailoring to specific needs.

However, it's not a one-size-fits-all solution. Each use case requires meticulous evaluation given the potential increase in architectural complexity and operational demands. But when approached strategically, the benefits of microservices can significantly outweigh these challenges. The key is in proactive planning, especially in areas of observability, security, and change management.

It's also important to note that beyond microservices, there are entirely different architectural frameworks like Generative AI architectures such as Retrieval Augmented Generation (RAG), providing a range of options to best fit your needs.

AWS, with its robust suite of managed services, empowers teams to build efficient microservices architectures and effectively minimize complexity. This whitepaper has aimed to guide you through the relevant AWS services and the implementation of key patterns. The goal is to equip you with the knowledge to harness the power of microservices on AWS, enabling you to capitalize on their benefits and transform your application development journey.

Contributors

The following individuals and organizations contributed to this document:

- Sascha Möllering, Solutions Architecture, Amazon Web Services
- Christian Müller, Solutions Architecture, Amazon Web Services
- Matthias Jung, Solutions Architecture, Amazon Web Services
- Peter Dalbhanjan, Solutions Architecture, Amazon Web Services
- Peter Chapman, Solutions Architecture, Amazon Web Services
- Christoph Kassen, Solutions Architecture, Amazon Web Services
- Umair Ishaq, Solutions Architecture, Amazon Web Services
- Rajiv Kumar, Solutions Architecture, Amazon Web Services
- Ramesh Dwarakanath, Solutions Architecture, Amazon Web Services
- Andrew Watkins, Solutions Architecture, Amazon Web Services
- Yann Stoneman, Solutions Architecture, Amazon Web Services
- Mainak Chaudhuri, Solutions Architecture, Amazon Web Services

Document history

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Major update	Added information about AWS Customer Carbon Footprint Tool, Amazon EventBridge, AWS AppSync (GraphQL), AWS Lambda Layers, Lambda SnapStart , Large Language Models (LLMs), Amazon Managed Streaming for Apache Kafka (MSK), Amazon Managed Workflows for Apache Airflow (MWAA), Amazon VPC Lattice, AWS AppConfig. Added separate section on cost optimization and sustainab ility.	July 31, 2023
Minor updates	Added Well-Architected to abstract.	April 13, 2022
Whitepaper updated	Integration of Amazon EventBridge, AWS OpenTelem etry, AMP, AMG, Container Insights, minor text changes.	November 9, 2021
Minor updates	Adjusted page layout	April 30, 2021
Minor updates	Minor text changes.	August 1, 2019
Whitepaper updated	Integration of Amazon EKS, AWS Fargate, Amazon MQ,	June 1, 2019

	AWS PrivateLink, AWS App Mesh, AWS Cloud Map	
Whitepaper updated	Integration of AWS Step Functions, AWS X-Ray, and ECS event streams.	September 1, 2017
Initial publication	Implementing Microservices on AWS published.	December 1, 2016

Note

To subscribe to RSS updates, you must have an RSS plug-in enabled for the browser you are using.

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.