Microservices on AWS

September 2017

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

Microservices are an architectural and organizational approach to software development designed to speed up deployment cycles, foster innovation and ownership, and improve maintainability and scalability of software applications. This approach includes scaling organizations that deliver software and services. Using a microservices approach, software is composed of small independent services that communicate over well-defined APIs. These services are owned by small self-contained teams.

In this whitepaper, we summarize the common characteristics of microservices, talk about the main challenges of building microservices, and describe how product teams can leverage Amazon Web Services (AWS) to overcome those challenges.

Introduction

For the last several years, microservices have been an important trend in IT architecture.[1] Microservices architectures are not a completely new approach to software engineering, but rather they are a collection and combination of successful and proven concepts such as agile software development, service-oriented architectures, API-first design, and Continuous Delivery (CD). In many cases, design patterns of the Twelve-Factor App are leveraged for microservices.[2]

Characteristics of Microservices

Microservices includes so many concepts that it is challenging to define it precisely. However, all microservices architectures share some common characteristics, as Figure 1 illustrates:

- **Decentralized** – Microservices architectures are distributed systems with decentralized data management. They don't rely on a unifying schema in a central database. Each microservice has its own view on data models. Microservices are also decentralized in the way they are developed, deployed, managed, and operated.
- **Independent** – Different components in a microservices architecture can be changed, upgraded, or replaced independently without affecting the functioning of other components. Similarly, the teams responsible for different microservices are enabled to act independently from each other.
- **Do one thing well** – Each microservice component is designed for a set of capabilities and focuses on a specific domain. If developers contribute so much code to a particular component of a service that the component reaches a certain level of complexity, then the service could be split into two or more services.
- **Polyglot** – Microservices architectures don't follow a "one size fits all" approach. Teams have the freedom to choose the best tool for their specific problems. As a consequence, microservices architectures take a heterogeneous approach to operating systems, programming languages, data stores, and tools. This approach is called polyglot persistence and programming.

- **Black box** – Individual microservice components are designed as black boxes, that is, they hide the details of their complexity from other components. Any communication between services happens via well-defined APIs to prevent implicit and hidden dependencies.
- **You build it; you run it** – Typically, the team responsible for building a service is also responsible for operating and maintaining it in production. This principle is also known as DevOps.[3] DevOps also helps bring developers into close contact with the actual users of their software and improves their understanding of the customers' needs and expectations. The fact that DevOps is a key organizational principle for microservices shouldn't be underestimated because according to Conway's law, system design is largely influenced by the organizational structure of the teams that build the system.[4]

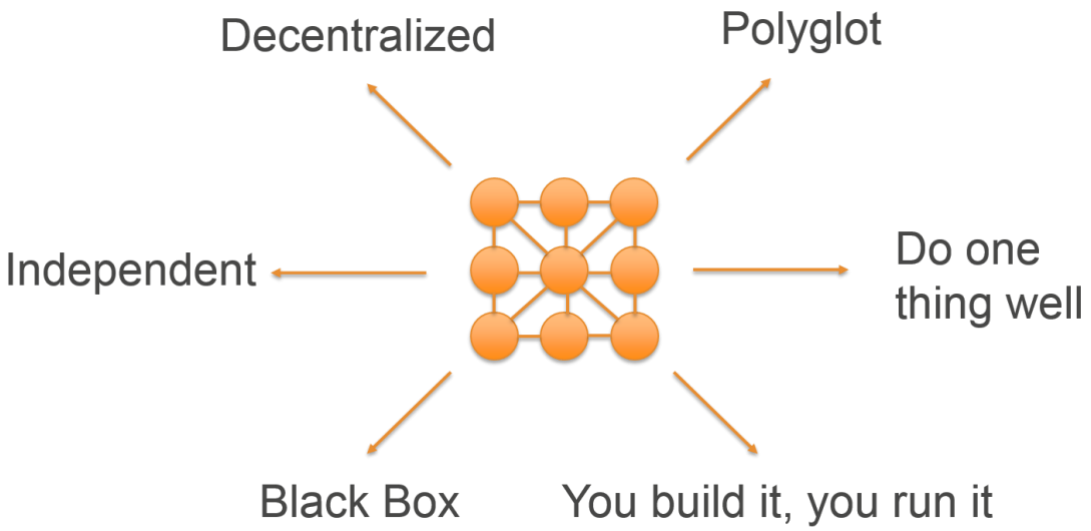

Figure 1: Characteristics of microservices

Benefits of Microservices

Many AWS customers adopt microservices to address limitations and challenges with agility and scalability that they experience in traditional monolithic deployments. Let's look at the main drivers for choosing a microservices architecture.

Agility

Microservices foster an organization of small independent teams that take ownership of their services. Teams act within a small and well-understood bounded context, and they are empowered to work independently and quickly, thus shortening cycle times. You benefit significantly from the aggregate throughput of the organization.

Figure 2 illustrates two types of deployment structures: many small independent teams working on many deployments versus a single large team working on a monolithic deployment.

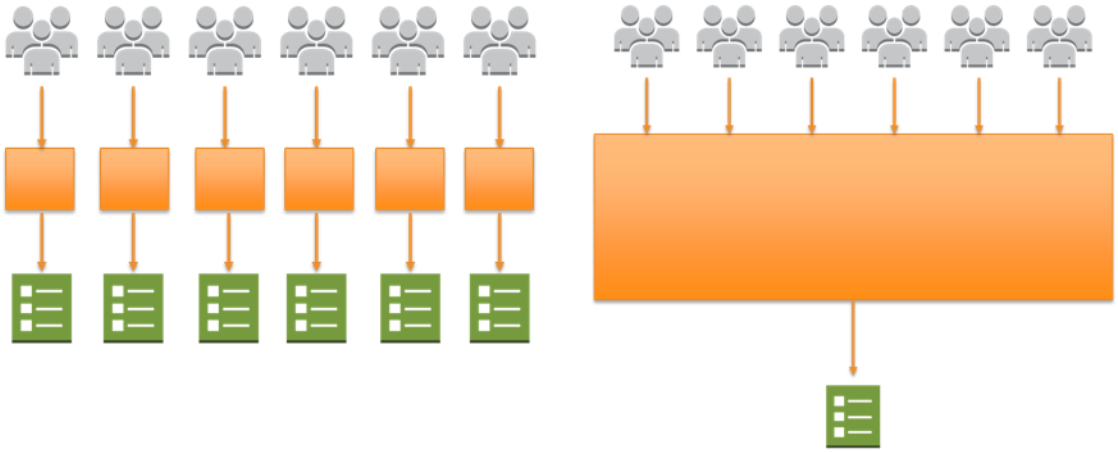

Figure 2: Deploying microservices

Innovation

The fact that small teams can act autonomously and choose the appropriate technologies, frameworks, and tools for their domains is an important driver for innovation. Responsibility and accountability foster a culture of ownership for services.

Establishing a DevOps culture by merging development and operational skills in the same group eliminates possible frictions and contradicting goals. Agile processes no longer stop when it comes to deployment. Instead, the complete application life-cycle management processes—from committing to running code—can be automated as a Continuous Delivery process. It becomes easy to test new ideas quickly and to roll back in case something doesn't work. The low cost of failure creates a culture of change and innovation.

Quality

Organizing software engineering around microservices can also improve the quality of code. The benefits of dividing software into small and well-defined modules are similar to those of object-oriented software engineering: improved reusability, composability, and maintainability of code.

Scalability

Fine-grained decoupling of microservices is a best practice for building large-scale systems. It's a prerequisite for performance optimization since it allows choosing the appropriate and optimal technologies for a specific service. Each service can be implemented with the appropriate programming languages and frameworks, leverage the optimal data persistence solution, and be fine-tuned with the best performing service configurations.

Properly decoupled services can be scaled horizontally and independently from each other. *Vertical scaling*, which is running the same software on bigger machines, is limited by the capacity of individual servers and can incur downtime during the scaling process. *Horizontal scaling*, which is adding more servers to the existing pool, is highly dynamic and doesn't run into limitations of individual servers. The scaling process can be completely automated. Furthermore, resiliency of the application can be improved because failing components can be easily and automatically replaced.

Availability

Microservices architectures make it easier to implement failure isolation. Techniques such as health-checking, caching, bulkheads, or circuit breakers allow you to reduce the blast radius of a failing component and to improve the overall availability of a given application.

Challenges of Microservices

Despite all the advantages that we have discussed, you should be aware that—as with all architectural styles—a microservices approach is not without its challenges. This section discusses some of the problems and trade-offs of the microservices approach.[5]

- **Distributed Systems** - Microservices are effectively a distributed system, which presents a set of problems often referred to as the

Fallacies of Distributed Computing.[6] Programmers new to distributed systems often assume the network to be reliable, the latency zero, and the bandwidth to be infinite.

- **Migration**- The migration process from a monolithic architecture to a microservices architecture requires you to determine the right boundaries for microservices. This process is complex and requires you to disentangle code dependencies going down to the database layer.
- **Versions** - Versioning for microservices can be challenging. There are several best practices and patterns, for example, routing-based versioning, which can be applied at the API level.
- **Organization** - Microservices architecture and organization architecture go hand in hand. Organizational problems include how to: build an effective team structure, transform the organization to follow a DevOps approach, and streamline communication between development and operations.

In this whitepaper, we mainly focus on the architectural and operational challenges of a move to microservices. To learn more about DevOps and AWS, see https://aws.amazon.com/devops/.[7]

Architectural Complexity

In monolithic architectures, the complexity and the number of dependencies reside inside the code base, while in microservices architectures complexity moves to the interactions of the individual services that implement a specific domain (Figure 3).

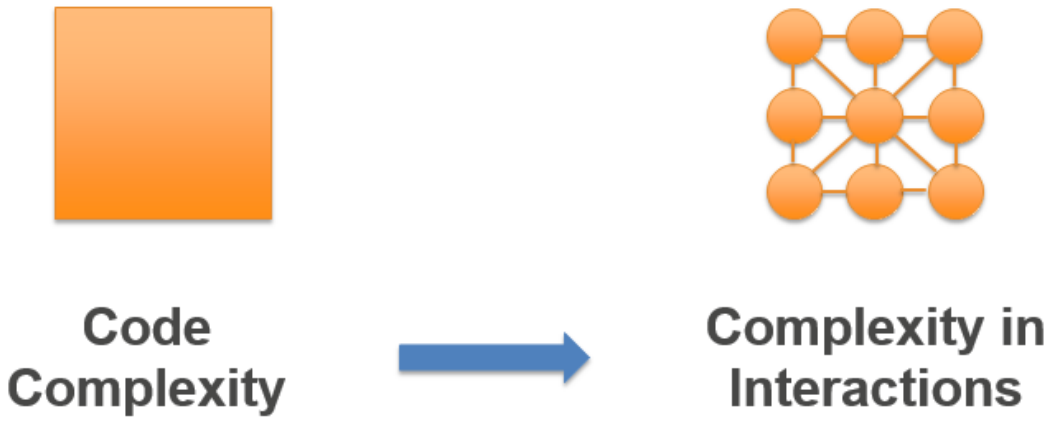

Figure 3: Complexity moves to interactions of individual microservices

Architectural challenges like dealing with asynchronous communication, cascading failures, data consistency problems, discovery, and authentication of services are critical to successful microservices implementation, and we'll address them in this paper.

Operational Complexity

With a microservices approach, you no longer run a single service, but dozens or even hundreds of services. This raises several questions:

- How to provision resources in a scalable and cost-efficient way?
- How to operate dozens or hundreds of microservice components effectively without multiplying efforts?
- How to avoid reinventing the wheel across different teams and duplicating tools and processes?
- How to keep track of hundreds of pipelines of code deployments and their interdependencies?
- How to monitor overall system health and identify potential hotspots early on?
- How to track and debug interactions across the whole system?
- How to analyze high amounts of log data in a distributed application that quickly grows and scales beyond anticipated demand?

- How to deal with a lack of standards and heterogeneous environments that include different technologies and people with differing skill sets?
- How to value diversity without locking into a multiplicity of different technologies that need to be maintained and upgraded over time?
- How to deal with versioning?
- How to ensure that services are still in use especially if the usage pattern isn't consistent?
- How to ensure the proper level of decoupling and communication between services?

Microservices and the Cloud

AWS has a number of offerings that address the most important challenges of microservices architectures:

- **On-demand resources** – AWS resources are available and rapidly provisioned when needed. Compared to traditional infrastructures, there is no practical limit on resources. Different environments and versions of services can temporarily or persistently co-exist. There is no need for difficult forecasting and guessing capacity. On-demand resources address the challenge of provisioning and scaling resources in a cost-efficient way.
- **Experiment with low cost and risk** – The fact that you only pay for what you use dramatically reduces the cost of experimenting with new ideas. New features or services can be rolled out easily and shut down again if they aren't successful. Reducing cost and risk for experimenting with new ideas is a key element of driving innovation. This perfectly fits with the goal of microservices to achieve high agility.
- **Programmability** – AWS services come with an API, Command Line Interface (CLI), and an SDK for different programming languages. Servers or even complete architectures can be programmatically cloned, shut down, scaled, and monitored. Additionally, in case of failure, they can heal themselves automatically. Standardization and automation are keys to building speed, consistency, repeatability, and scalability. You are empowered to summon the resources you need through code and build-dedicated tools to minimize operational efforts for running microservices.

- **Infrastructure as code** – In addition to using programmatic scripts to provision and manage an infrastructure, AWS allows you to describe the whole infrastructure as code and manage it in a version control system–just as you do for application code. As a consequence, any specific version of an infrastructure can be redeployed at any time. You can compare the quality and performance of a specific infrastructure version with a specific application version and ensure that they are in sync. Rollbacks are no longer limited to the application—they can include the whole infrastructure.[8]
- **Continuous Delivery** – The programmability of the cloud allows automation of the provisioning and deployment process. Continuous Integration within the development part of the application lifecycle can be extended to the operations part of the lifecycle. This enables the adoption of Continuous Deployment and Delivery.[9, 10] Continuous delivery addresses the challenges of operational complexity that occur when you manage multiple application life cycles in parallel.
- **Managed services** – A key benefit of cloud infrastructures is managed services. Managed services relieve you of the heavy lifting of provisioning virtual servers, installing, configuring and optimizing software, dealing with scaling and resilience, and doing reliable backups. System characteristics and features such as monitoring, security, logging, scalability, and availability are already built into those services. Managed services are a major element you can use to reduce the operational complexity of running microservices architectures.
- **Service orientation** – AWS itself follows a service-oriented structure. Each AWS service focuses on solving a well-defined problem and communicates with other services using clearly defined APIs. You can put together complex infrastructure solutions by combining those service primitives like LEGO blocks. This approach prevents reinventing the wheel and the duplication of processes.
- **Polyglot** – AWS provides a large choice of different storage and database technologies. Many popular operating systems that run on Amazon Elastic Compute Cloud (Amazon EC2) are available on the AWS Marketplace.[11] In addition AWS supports a large variety of programming languages with SDKs.[12] This enables you to use the most appropriate solution for your specific problem.

Microservices on AWS

In this section, 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. Next, we recommend the AWS services that are best for implementing a typical serverless microservices architecture that reduces operational complexity. 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

In the past, typical monolithic applications were built using different layers, for example, 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 4 depicts a reference architecture for a typical microservices application on AWS.

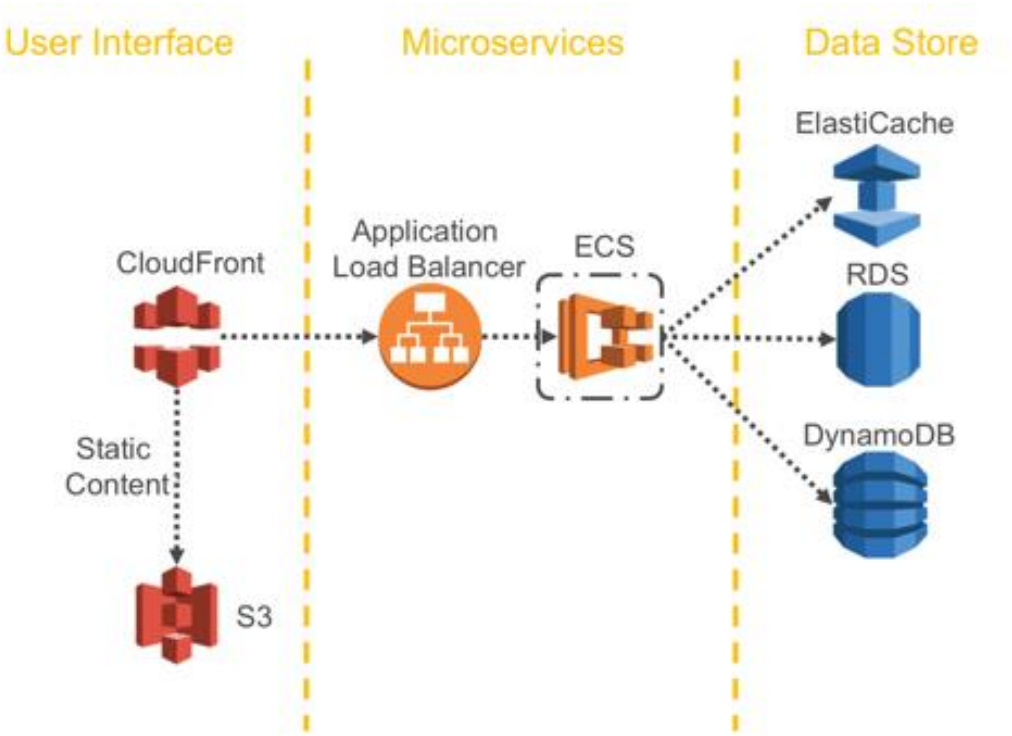

Figure 4: Typical microservices application on AWS

User Interface

Modern web applications often use JavaScript frameworks to implement a single-page application that communicates with a RESTful API. Static web content can be served using Amazon Simple Storage Service (Amazon S3) and Amazon CloudFront.

> CloudFront is a global content delivery network (CDN) service that accelerates delivery of your websites, APIs, video content, and other web assets.[13]

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 even add more latency. It is a best practice to implement other caching mechanisms to reduce chattiness and minimize latencies.

Microservices

The API of a microservice is the central entry point for all client requests. The application logic hides behind a set of programmatic interfaces, typically a RESTful web services API.[14] This API accepts and processes calls from clients and might implement functionality such as traffic management, request filtering, routing, caching, and authentication and authorization.

Many AWS customers use the Elastic Load Balancing (ELB) Application Load Balancer together with Amazon EC2 Container Service (Amazon ECS) and Auto Scaling to implement a microservices application. The Application Load Balancer routes traffic based on advanced application-level information that includes the content of the request.

> ELB automatically distributes incoming application traffic across multiple Amazon EC2 instances.[15]

The Application Load Balancer distributes incoming requests to Amazon ECS container instances running the API and the business logic.

> Amazon EC2 is a web service that provides secure, resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers.[16]
>
> Amazon EC2 Container Service (Amazon ECS) is a highly scalable, high performance container management service that supports Docker containers and allows you to easily run applications on a managed cluster of Amazon EC2 instances.[17]

Amazon ECS container instances are scaled out and scaled in, depending on the load or the number of incoming requests. Elastic scaling allows the system to be run in a cost-efficient way and also helps protect against denial of service attacks.

> Auto Scaling helps you maintain application availability and allows you to scale your Amazon EC2 capacity up or down automatically according to conditions you define.[18]

Containers

A common approach to reducing operational efforts for deployment is container-based deployment. Container technologies like Docker [19] have increased in popularity in the last few years due to the following benefits:

- **Portability –** Container images are consistent and immutable, that is, they behave the same no matter where they are run (on a developer's desktop as well as in a production environment).
- **Productivity –** Containers increase developer productivity by removing cross-service dependencies and conflicts. Each application component can be broken into different containers running a different microservice.
- **Efficiency –** Containers allow the explicit specification of resource requirements (CPU, RAM), which makes it easy to distribute containers across underlying hosts and significantly improve resource usage. Containers also have only a light performance overhead compared to

virtualized servers and efficiently share resources on the underlying operating system.

- **Control** – Containers automatically version your application code and its dependencies. Docker container images and Amazon ECS task definitions allow you to easily maintain and track versions of a container, inspect differences between versions, and roll back to previous versions.

Amazon ECS eliminates 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 balancers, Amazon Elastic Block Store (Amazon EBS) volumes, and AWS Identity and Access Management (IAM) roles.

After a cluster of EC2 instances is up and running, you can define task definitions and services that specify which Docker container images to run on the cluster. Container images are stored in and pulled from container registries, which may exist within or outside your AWS infrastructure. To define how your applications run on Amazon ECS, you create a task definition in JSON format. This task definition defines parameters for which container image to run, CPU, memory needed to run the image, how many containers to run, and strategies for container placement within the cluster. Other parameters include security, networking, and logging for your containers.

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.

Docker images used in Amazon ECS can be stored in Amazon EC2 Container Registry (Amazon ECR). Amazon ECR[20] eliminates the need to operate and scale the infrastructure required to power your container registry.

> Amazon EC2 Container Registry (Amazon ECR) is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images. Amazon ECR is integrated with Amazon EC2 Container Service (Amazon ECS), simplifying your development to production workflow.

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.

> Amazon ElastiCache is a web service that makes it easy to deploy, operate, and scale an in-memory data store or cache in the cloud.[21] The service improves the performance of web applications by allowing you to retrieve information from fast, managed, in-memory caches, instead of relying entirely on slower disk-based databases.

Putting a cache between application servers and a database is a common mechanism to alleviate read load from 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 for storing structured data and business objects. AWS offers six database engines (Microsoft SQL Server, Oracle, MySQL, MariaDB, PostgreSQL, and Amazon Aurora) as managed services via Amazon Relational Database Service (Amazon RDS).

> Amazon RDS makes it easy to set up, operate, and scale a relational database in the cloud.[22] It provides cost-efficient and resizable capacity while managing time-consuming database administration tasks, freeing you to focus on applications and business.

Relational databases, however, are not designed for endless scale, which can make it very hard 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 is that NoSQL databases typically do not enforce a strict schema. Data is distributed over partitions that can be scaled horizontally and is retrieved via partition keys.

Since 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.

> Amazon DynamoDB is a fast and flexible NoSQL database service for all applications that need consistent, single-digit millisecond latency at any scale.[23]

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 automatically spreads the data and traffic for the table over a sufficient number of servers to handle the request capacity specified by the customer and the amount of data stored, while maintaining consistent and fast performance.

DynamoDB is designed for scale and performance. In most cases, DynamoDB response times can be measured in single-digit milliseconds. However, there are certain use cases that require response times in microseconds. For these use cases, DynamoDB Accelerator (DAX) provides caching capabilities for accessing eventually consistent data. DAX does all the heavy lifting required to add in-memory acceleration to your DynamoDB tables, without requiring developers to manage cache invalidation, data population, or cluster management.

DynamoDB provides an auto scaling feature to dynamically adjust provisioned throughput capacity on your behalf, in response to actual traffic patterns. *Provisioned throughput* is the maximum amount of capacity that an application can consume from a table or index. When the workload decreases, Application

Auto Scaling decreases the throughput so that you don't pay for unused provisioned capacity.

Reducing Operational Complexity

The architecture we have described is already using managed services, but you still have to operate EC2 instances. We can further reduce the operational efforts needed to run, maintain, and monitor microservices by using a fully serverless architecture.

API Implementation

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

Access 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 API.

Other important features and challenges include throttling requests to protect the backend, caching API responses, request and response transformation, and generating API definitions and documentation with tools such as Swagger.[24]

Amazon API Gateway addresses those challenges and reduces the operational complexity of creating and maintaining RESTful APIs.

> API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale.[25]

API Gateway allows you to create your APIs programmatically by importing Swagger definitions by using the AWS API or by using the AWS Management Console. API Gateway serves as a front door to any web application running on Amazon EC2, Amazon ECS, AWS Lambda, or on any on-premises environment. In a nutshell: It allows you to run APIs without managing servers.

Figure 5 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. Additionally, CloudFront offers Regional Edge Caches. These locations are deployed globally at close proximity to your viewers. They sit between your origin server and the global edge locations that serve traffic directly to your viewers. API Gateway first checks if the request is in the cache at either an edge location or Regional Edge Cache location and, if no cached records are available, then forwards it to the backend for processing. This only applies to GET requests—all other request methods are automatically passed through. After the backend has processed the request, API call metrics are logged in Amazon CloudWatch, and content is returned to the client.

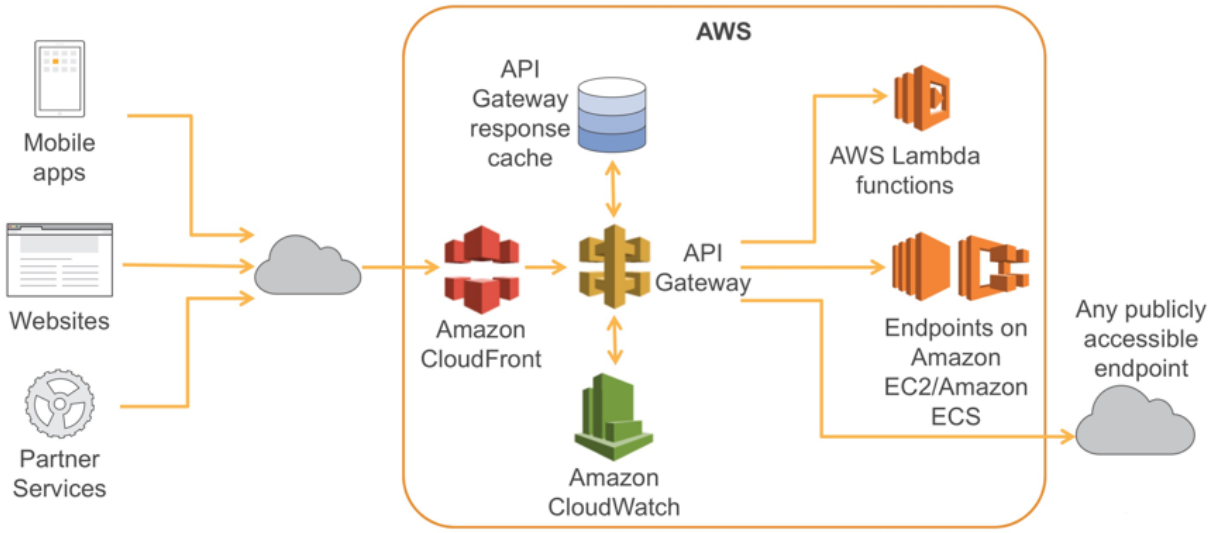

Figure 5: API Gateway call flow

Serverless Microservices

"No server is easier to manage than no server".[26] Getting rid of servers is the ultimate way to eliminate operational complexity.

AWS Lambda lets you run code without provisioning or managing servers.[27] You pay only for the compute time you consume – there is no charge when your code is not running. With Lambda, you can run code

for virtually any type of application or backend service – all with zero administration.

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. Lambda supports several programming languages and can be triggered from other AWS services or be called directly from any web or mobile application.

Lambda is highly integrated with API Gateway. The possibility of making synchronous calls from API Gateway to AWS Lambda enables the creation of fully serverless applications and is described in detail in our documentation.[28]

Figure 6 shows the architecture of a serverless microservice where the complete service is built out of managed services. This eliminates the architectural burden of designing for scale and high availability and eliminates the operational efforts of running and monitoring the microservice's underlying infrastructure.

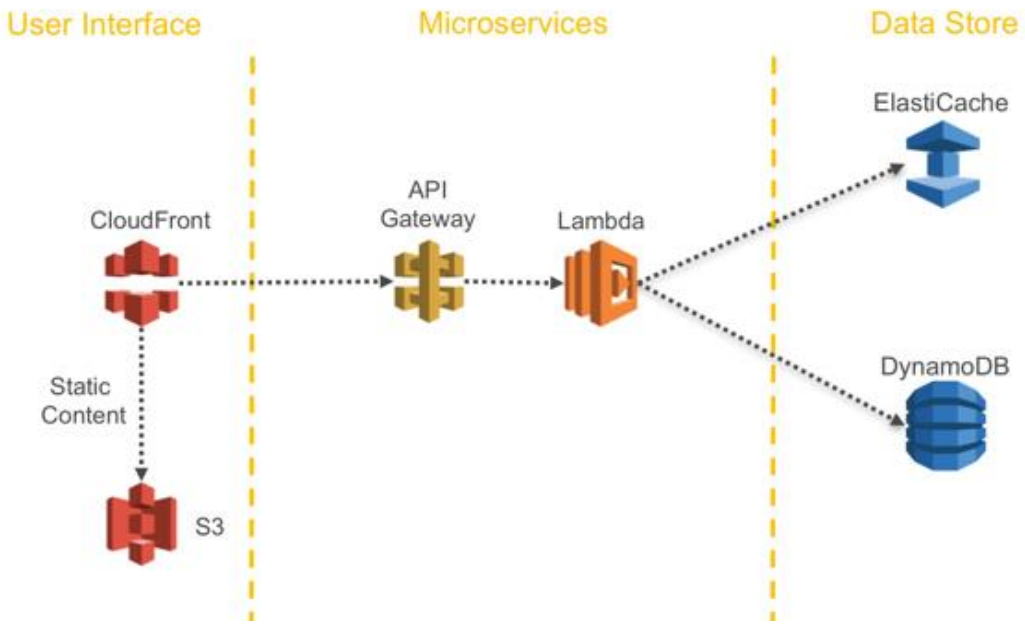

Figure 6: Serverless microservice

Deploying Lambda-Based Applications

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

> CloudFormation is a service that helps you model and set up your AWS resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS.[29]

The AWS Serverless Application Model (AWS SAM) is a convenient way to define serverless applications.[30] AWS SAM is natively supported by CloudFormation and defines a simplified syntax for expressing serverless resources. 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.

Distributed Systems Components

After looking at how AWS can solve challenges related to individual microservices, we now want to look at 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 they also present challenges, such as checking the health of those systems and announcing when new applications come online. In addition, you must decide how and where to store meta-store information, such as configuration data that can be used by applications. Here we explore several techniques for performing service discovery on AWS for microservices-based architectures.

Client-Side Service Discovery

The most simplistic approach for connecting different tiers or services is to hardcode the IP address of the target as part of the configuration of the

communication source. This configuration can be stored in Domain Name System (DNS) or application configuration and leveraged whenever systems need to communicate with each other. Obviously, this solution doesn't work well when your application scales. It isn't recommended for microservices architectures due to the dynamic nature of target properties. Every time the target system changes its properties—regardless of whether it's the IP address or port information—the source system has to update the configuration.

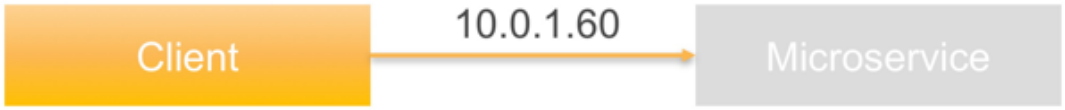

Figure 7: Client-side service discovery

Application Load Balancer-Based Service Discovery

One of the advantages of Application Load Balancing is that it provides health checks and automatic registration/de-registration of backend services in failure cases. The Application Load Balancer also offers path- and host-based routing approaches. Combining these features with DNS capabilities, it's possible to build a simple service discovery solution with minimum efforts and low cost.

You can configure a custom domain name for each microservice and associate the domain name with the Application Load Balancer's DNS name using a CNAME entry.[31] The DNS names of the service endpoints are then published across other applications that need access.

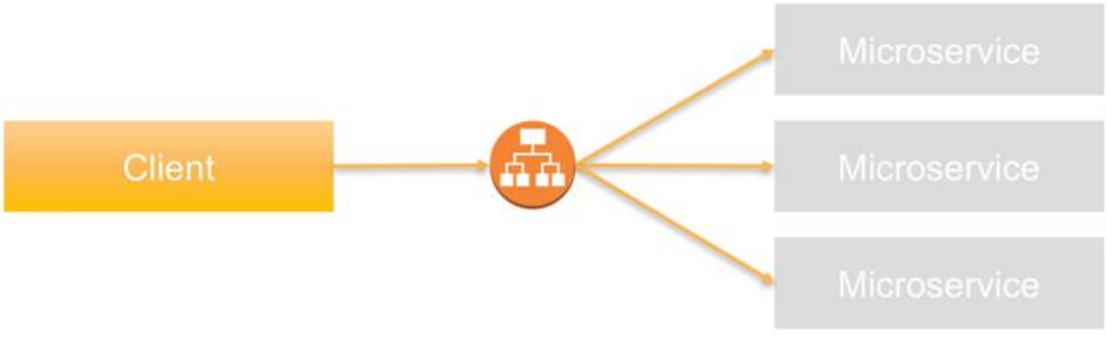

Figure 8: Application Load Balancer-based service discovery

DNS-Based Service Discovery

Amazon Route 53 could be another source for holding service discovery information.

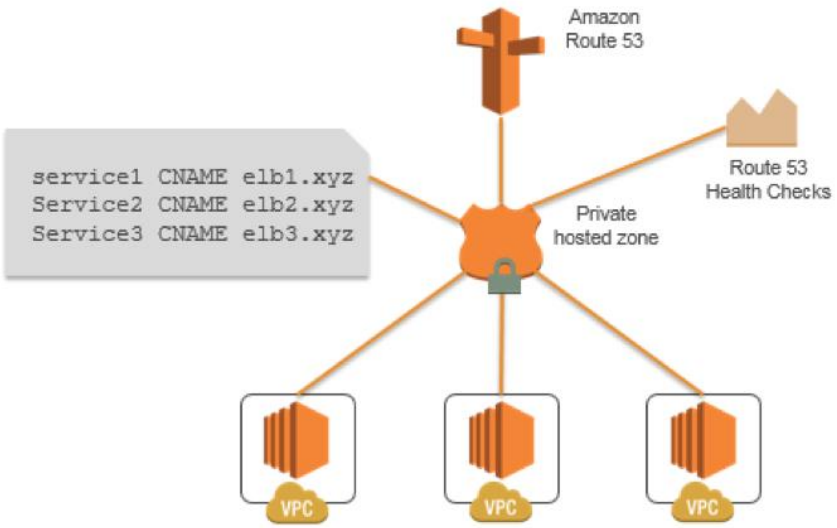

Figure 9: Domain Name System-based service discovery

> Route 53 is a highly available and scalable cloud DNS web service.[32]

Route 53 provides several features that can be leveraged for service discovery. The private hosted zones feature allows it to hold DNS record sets for a domain or subdomains and restrict access to specific virtual private clouds (VPCs).[33] You register IP addresses, hostnames, and port information as service records (SRV records) for a specific microservice and restrict access to the VPCs of the relevant client microservices. You can also configure health checks that regularly verify the status of the application and potentially trigger a failover among resource records.[34]

Service Discovery Using Amazon ECS Event Stream

A different approach to implementing Route 53-based service discovery is to leverage the capabilities of the Amazon ECS event stream feature.

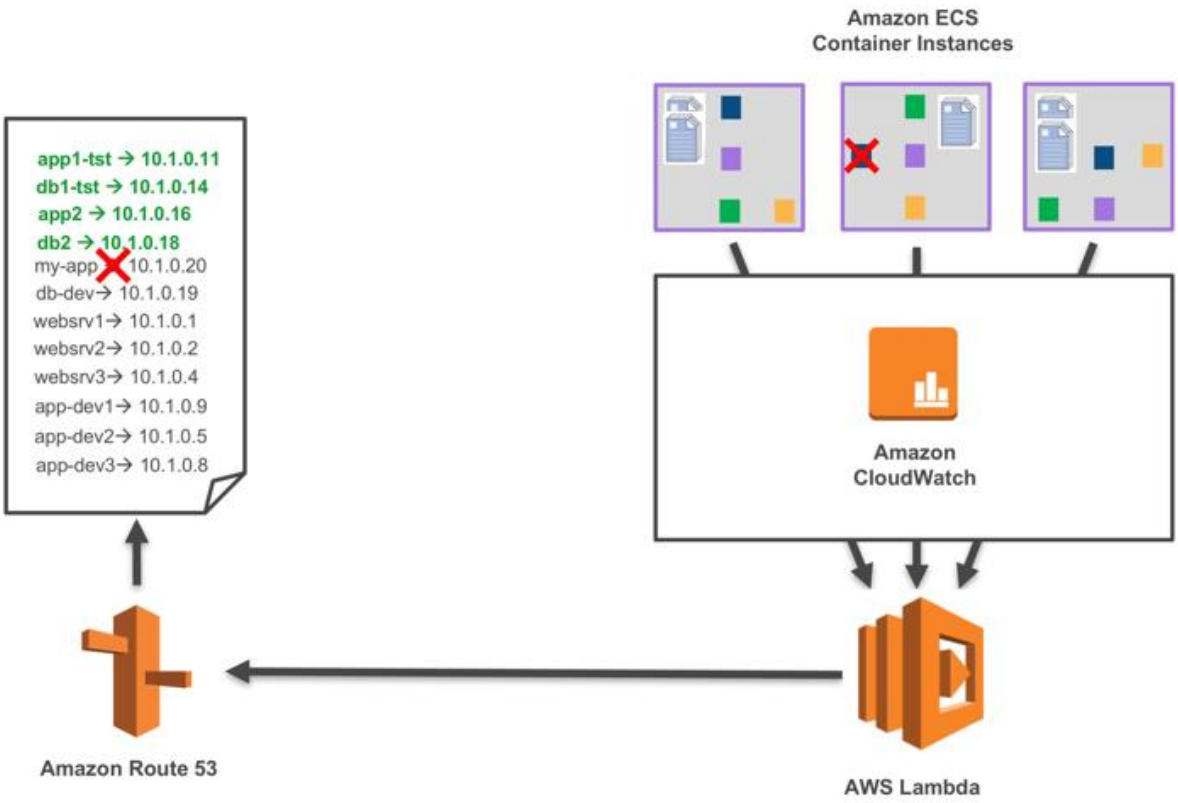

Figure 10: Service discovery using Amazon ECS event stream

You can use Amazon ECS event stream for CloudWatch events to receive near real-time notifications regarding the current state of both the container instances within an Amazon ECS cluster and the current state of all tasks running on those container instances. It is possible to use CloudWatch rules to filter on specific changes within the ECS cluster (e.g., start, stop) and use that information to update the DNS entries in Route 53.[35]

Service Discovery Using Configuration Management

Using Configuration Management tools (like Chef, Puppet, or Ansible) is another way to implement service discovery. Agents running on EC2 instances can register configuration information during server start. This information can be stored either on hosts or a centralized store along with other configuration management information.

One of the challenges of using configuration management tools is the frequency of updating health check information. Configuration of clients must be done thoroughly to retrieve the health of the application and to propagate updates immediately to prevent stale status information.

Figure 11 shows a service discovery mechanism using the configuration management system AWS OpsWorks.

> OpsWorks is a configuration management service that uses Chef, an automation platform that treats server configurations as code. OpsWorks uses Chef to automate how servers are configured, deployed, and managed across your EC2 instances or on-premises compute environments.[36]

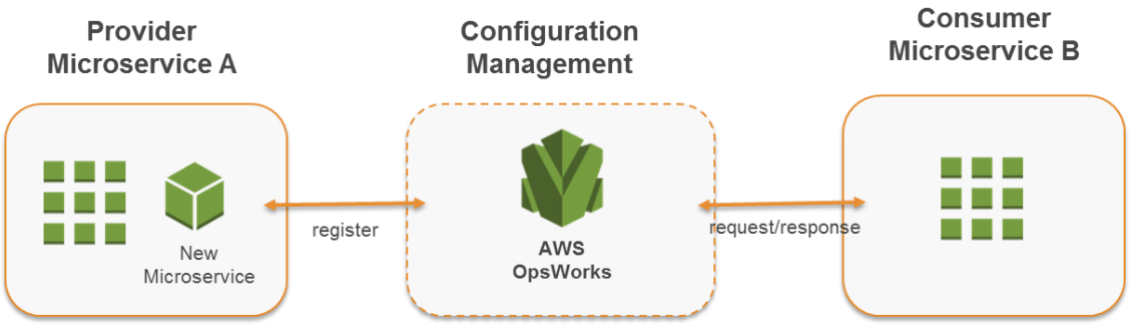

Figure 11: Service discovery using configuration management

Service Discovery Using Key Value Store

You can also use a key-value store for discovery of microservices. Although it takes longer to build this approach compared to other approaches, it provides more flexibility and extensibility and doesn't encounter DNS caching issues. It also works well with client-side load-balancing techniques such as Netflix Ribbon.[37] Client-side load balancing can help eliminate bottlenecks and simplify management.

Figure 12 shows an architecture that leverages Amazon DynamoDB as a key-value store and Amazon DynamoDB Streams[38] to propagate status changes to other microservices.

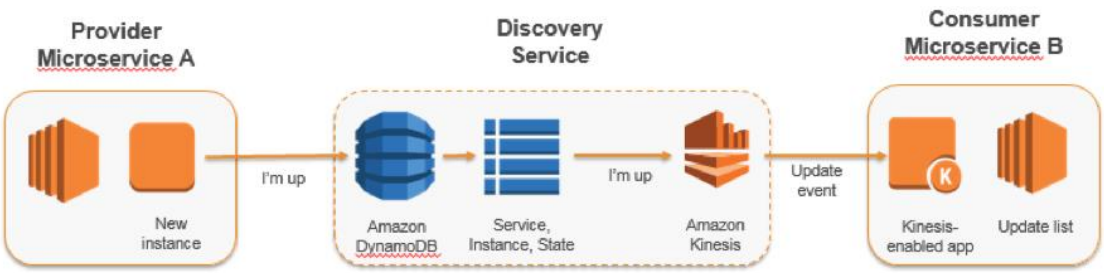

Figure 12: Service discovery using key/value store

Third-party software

A different approach to implementing service discovery is using third-party software like HashiCorp Consul,[39] etcd,[40] or Netflix Eureka.[41] All three examples are distributed, reliable key-value stores. For HashiCorp Consul, there is an AWS Quick Start[42] that sets up a flexible, scalable AWS Cloud environment and launches HashiCorp Consul automatically into a configuration of your choice.

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 explained by the CAP Theorem,[43] distributed microservices architectures inherently trade off consistency for performance and need to embrace eventual consistency.

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.[44] Using AWS Lambda with scheduled Amazon CloudWatch Events you can build a simple cleanup and deduplication mechanism.[45]

It's very common for state changes to affect more than a single microservice. In those cases, event sourcing has proven to be a useful pattern.[46] 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 pattern (Command, Query, Responsibility, Segregation) to decouple read from write workloads and optimize both for performance, scalability, and security.[47] In traditional data management systems, commands and queries are run against the same data repository.

Figure 13 shows how the event sourcing pattern can be implemented on AWS. Amazon Kinesis Streams serves as the main component of the central event store that captures application changes as events and persists them on Amazon S3.

> Kinesis Streams enables you to build custom applications that process or analyze streaming data for specialized needs.[48] Kinesis Streams can continuously capture and store terabytes of data per hour from hundreds of thousands of sources, such as website clickstreams, financial transactions, social media feeds, IT logs, and location-tracking events.

Figure 13 depicts three different microservices composed of Amazon API Gateway, Amazon EC2, 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 Streams. All microservices run their own Kinesis Streams application on a fleet of EC2 instances that read a copy of the message, filter it based on relevancy for the microservice, and possibly forward it for further processing.

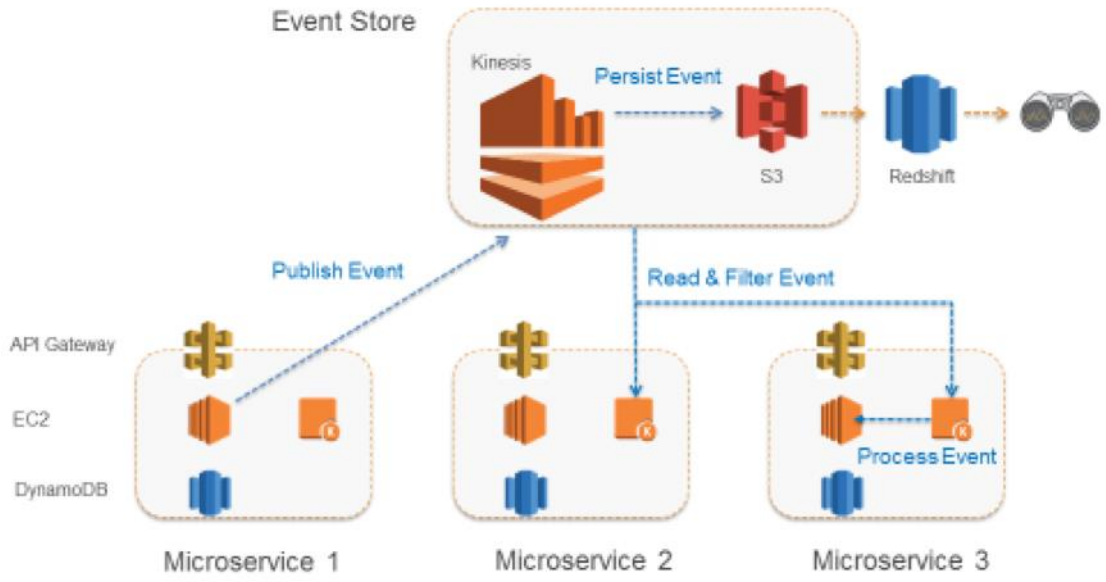

Figure 13: Event sourcing pattern on AWS

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

In traditional, monolithic applications communication is rather simple: parts of the application can communicate with other parts using method calls or an internal event distribution mechanism. If the same application is implemented using decoupled microservices, the communication between different parts of the application has to 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

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. Amazon Simple Queue Service (Amazon SQS) and Amazon Simple Notification Service (Amazon SNS) make it simple to implement this pattern.

> Amazon SQS is a fast, reliable, scalable, fully managed queuing service that makes it simple and cost effective to decouple the components of a cloud application.[49]
>
> Amazon SNS is fully managed notification service that provides developers with a highly scalable, flexible, and cost-effective capability to publish messages from an application and immediately deliver them to subscribers or other applications.[50]

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 14 demonstrates the integration of Amazon SNS and Amazon SQS.

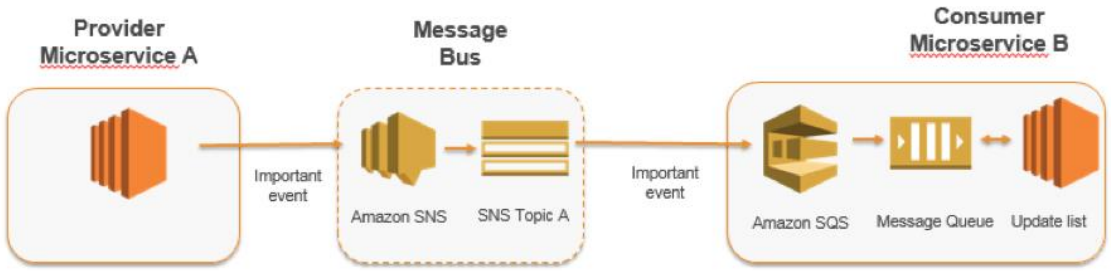

Figure 14: Message bus pattern on AWS

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.

Orchestration and State Management

The distributed character of microservices makes it challenging to orchestrate workflows with multiple microservices 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.

> AWS Step Functions makes it easy to coordinate the components of distributed applications and microservices using visual workflows.[51]

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. Figure 15 illustrates that invocations of Lambda functions are pushed directly from Step Functions to AWS 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 your application is available at any scale.

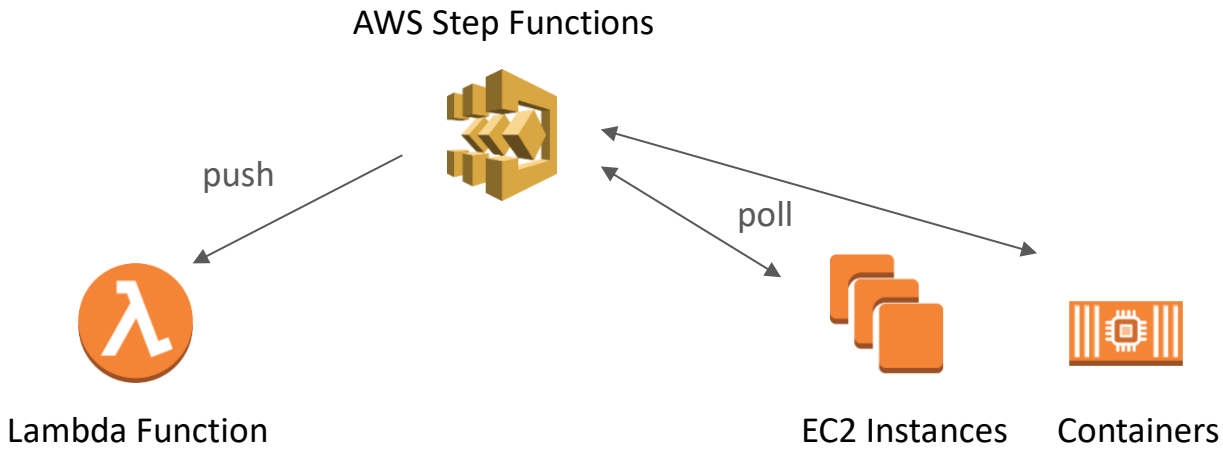

Figure 15: Orchestration with AWS Step Functions

To build workflows Step Functions uses the Amazon States Language.[52] Workflows can contain sequential or parallel steps as well as branching steps.

Figure 16 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.

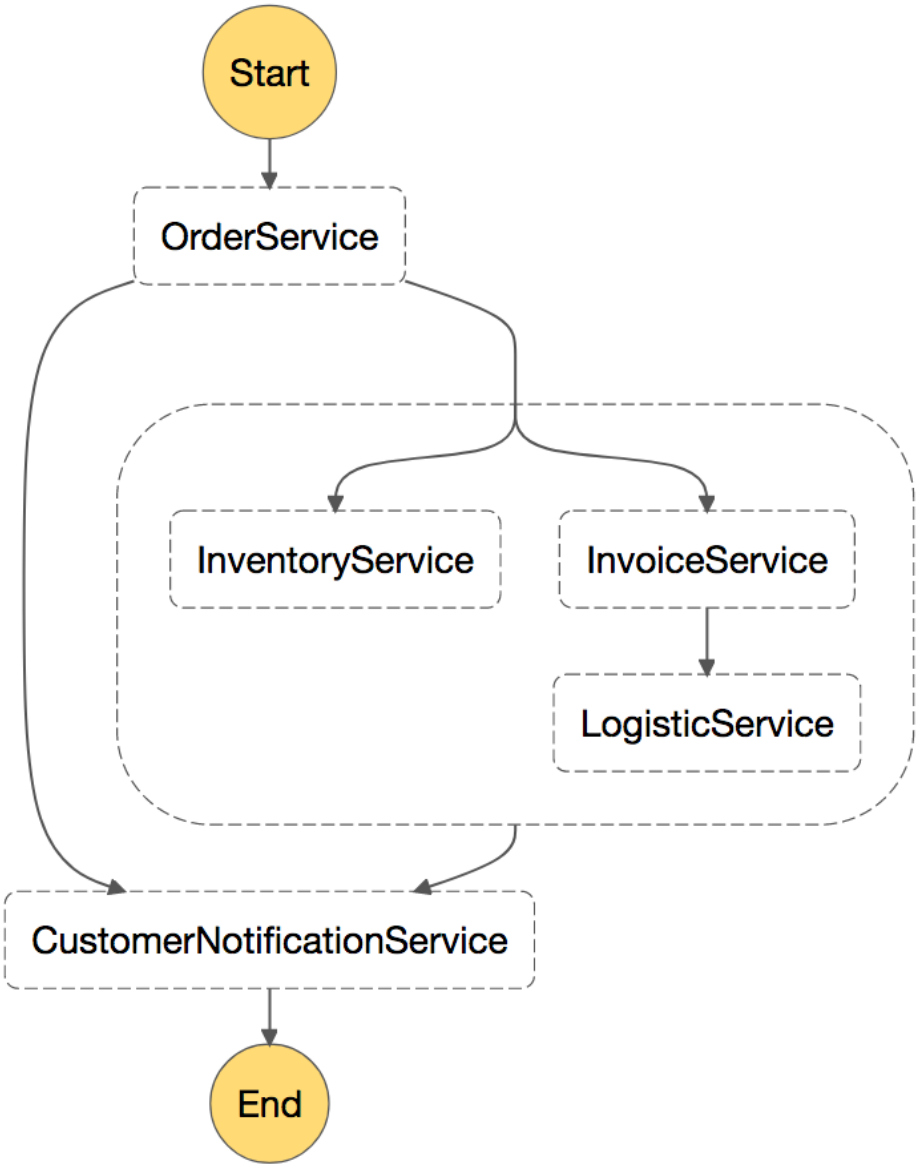

Figure 16: An example of a microservices workflow invoked by AWS Step Functions

Distributed Monitoring

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

> CloudWatch is a monitoring service for AWS Cloud resources and the applications you run on AWS.[53]

You can use CloudWatch to collect and track metrics, centralize and monitor log files, set alarms, and automatically react to changes in your AWS environment. CloudWatch can monitor AWS resources such as 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.[54]

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 already centralize log files. The primary destinations for log files on AWS are Amazon S3 and Amazon CloudWatch Logs. For applications running on top of EC2 instances a daemon is available to ship log files to CloudWatch Logs. Lambda functions natively ship 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.[55]

Figure 17 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) and Kibana. Amazon Athena can be used to run ad-hoc queries against centralized logfiles in Amazon S3.

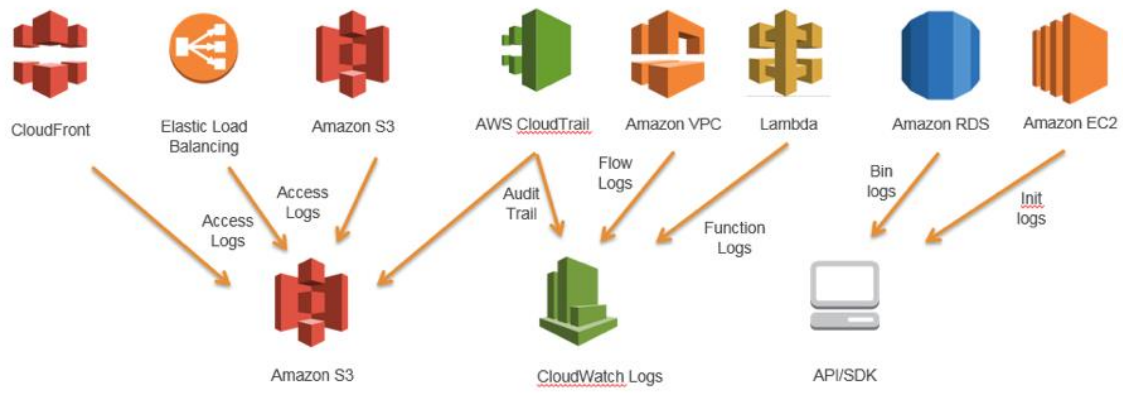

Figure 17: Logging capabilities of AWS services

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 very hard to find all relevant log messages.

> AWS X-Ray provides an end-to-end view of requests as they travel through your application and shows a map of your application's underlying components.[56]

The central idea behind X-Ray is the use of correlation IDs, which are unique identifiers attached to all requests and messages related to a specific event chain. The trace ID is added to HTTP requests in specific tracing headers named `X-Amzn-Trace-Id` when the request hits the first X-Ray-integrated service (for example, an Application Load Balancer or API Gateway) and is 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, AWS Lambda, and AWS Elastic Beanstalk. You can use X-Ray with applications written in Java, Node.js, and .NET that are deployed on these services.

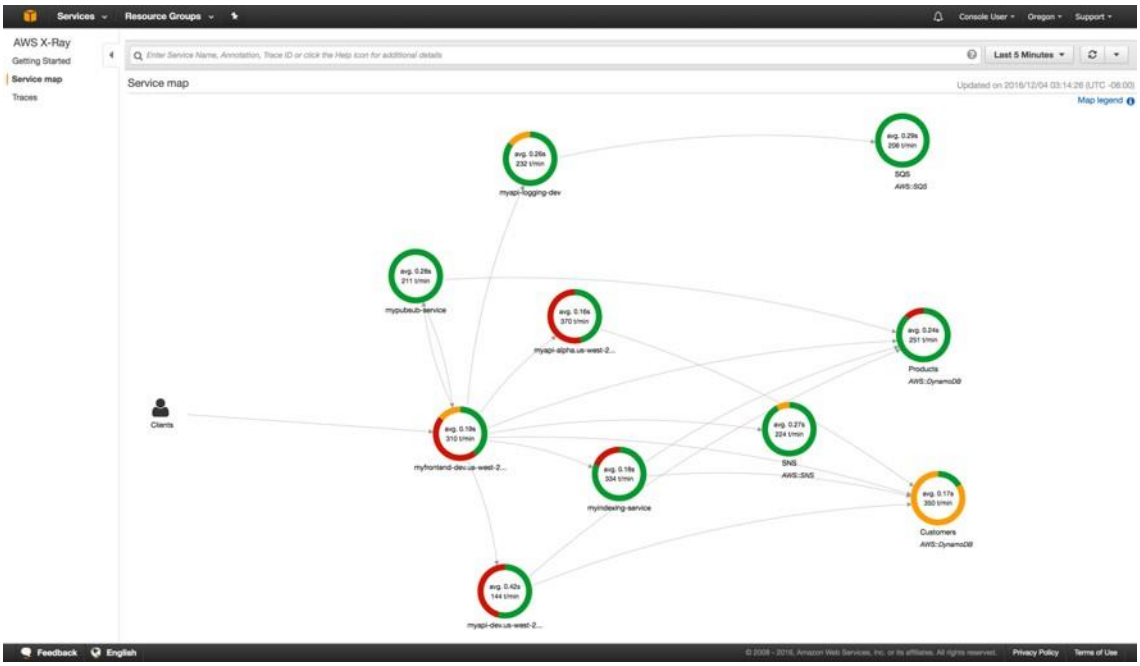

Figure 18: AWS X-Ray service map

Options for Log Analysis on AWS

Searching, analyzing, and visualizing log data is an important aspect of understanding distributed systems. One popular option for analyzing log files is to use Amazon ES together with Kibana.

> Amazon ES makes it easy to deploy, operate, and scale Elasticsearch for log analytics, application monitoring, interactive search, and more.[57]

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 19 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.[58]

Figure 19: Log analysis with Amazon Elasticsearch Service and Kibana

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

> Amazon Redshift is a fast, fully managed, petabyte-scale data warehouse service that makes it simple and cost-effective to analyze all your data using your existing business intelligence tools.[59]
>
> Amazon QuickSight is a fast, cloud-powered business analytics service to build visualizations, perform ad-hoc analysis, and quickly get business insights from your data.[60]

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 storing the data, it is possible to stream log entries to Amazon Kinesis Firehose.

> Kinesis Firehose is a fully managed service for delivering real-time streaming data to destinations such as Amazon S3, Amazon Redshift, or Amazon ES.[61]

Figure 20 depicts a scenario where log entries are streamed from different sources to Amazon Redshift using CloudWatch Logs and Kinesis Firehose. Amazon QuickSight uses the data stored in Amazon Redshift for analysis, reporting, and visualization.

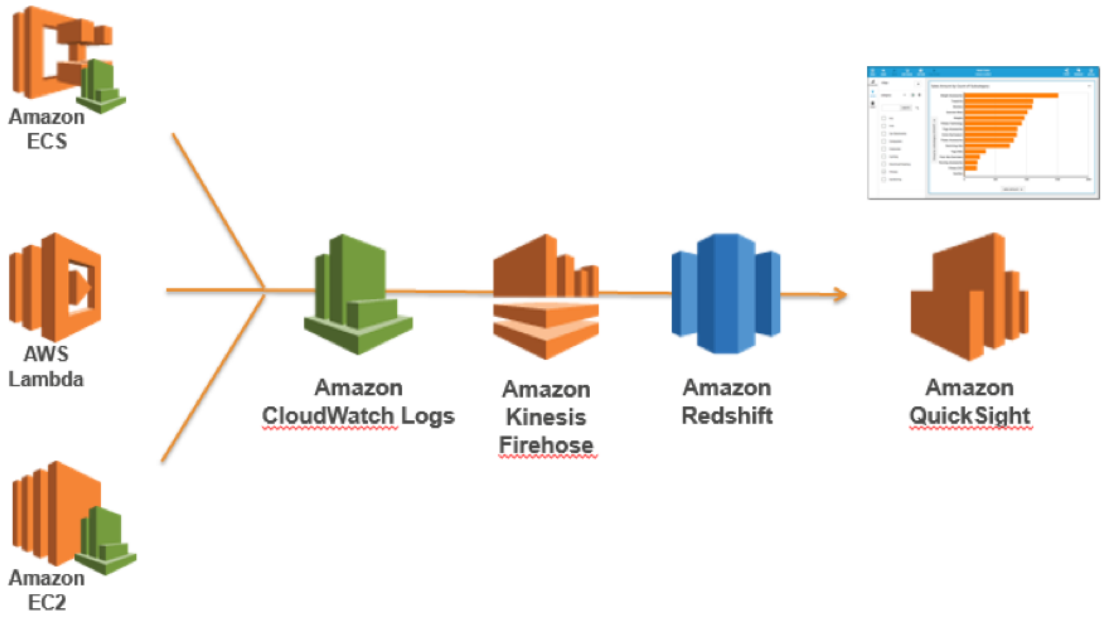

Figure 20: Log analysis with Amazon Redshift and Amazon QuickSight

Figure 21 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, for example, Amazon Redshift or Amazon EMR, to analyze the data stored in the log stream and find anomalies.

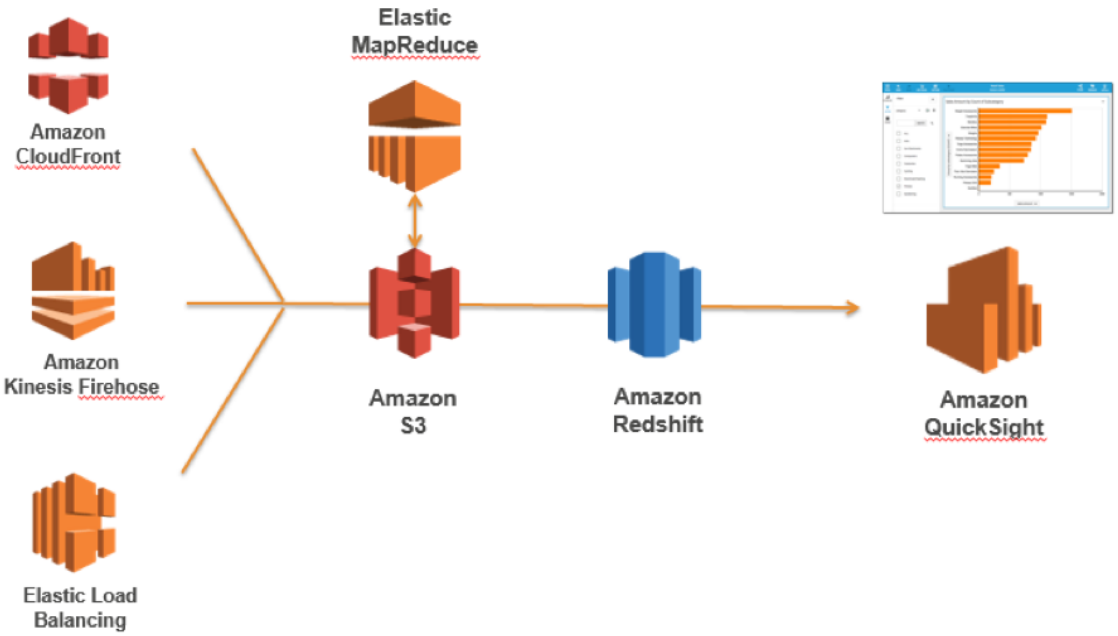

Figure 21: Log analysis on Amazon S3

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 as a communication protocol. It is a light-weight protocol, but high volumes can cause issues. In some cases, it might make sense to think about consolidating services that send a lot of 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 quite common to use simple protocols like HTTP. Messages exchanged by services can be encoded in different ways, for example, in a human-readable format like JSON or YAML or in an efficient binary format 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 amount 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 useful to reduce load from the data persistence layer. The challenge for all caching mechanisms 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 with potentially hundreds of distributed services is to ensure visibility of user actions on all services and to be able to get a good overall view at an organizational level. To help enforce security policies, it is important to audit both resource access as well as activities leading to system changes. Changes must be tracked on the level of individual services and on the wider system across services. It is typical in microservices architectures that changes happen very often, which means that auditing change becomes even more important. In this section, we look at the key services and features within AWS that can help audit your microservices architecture.

Audit Trail

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

> CloudTrail is a web service that records AWS API calls for your account and delivers log files to you.[62] This includes those taken on the AWS Management Console, the AWS CLI, SDKs, and calls made directly to the AWS API.

All user actions and automated systems become searchable and can be analyzed for unexpected behavior, company policy violations, or debugging. Information recorded includes user/account information, a timestamp, the service that was called along with the action 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]

Storing CloudTrail log files in S3 buckets has a few advantages: trail data is stored durably, new files can trigger an SNS notification or start a Lambda function to parse the log file, and data can be automatically archived into Amazon Glacier via lifecycle policies.[64] In addition (and as described earlier in the performance monitoring section), services like Amazon EMR or Amazon Redshift can be leveraged to further analyze the data.

The advantages of storing the audit trails in CloudWatch are that trail data is generated in real time and rerouting information to Amazon ES for search and visualization becomes very easy. It is possible to configure CloudTrail to log into both Amazon S3 and CloudWatch Logs.

Events and Real-Time Actions

There are certain changes in systems architectures that must be responded to quickly, and an action to remediate must be performed, or specific governance procedures to authorize must be followed.

> CloudWatch Events delivers a near real-time stream of system events that describe changes in AWS resources.[65] Declarative rules associate events of interest with automated actions to be taken.

The integration of CloudWatch Events with CloudTrail allows CloudWatch Events to generate events for all mutating API calls across all AWS services. It's also possible to define custom events or generate events based on a fixed schedule.

When an event is fired and matches a rule that you defined in your system, the right people in your organization can be immediately notified. This allows them to take the appropriate action. Even better, it's possible to automatically trigger built-in workflows or invoke a Lambda function.

Figure 22 shows a setup where CloudTrail and CloudWatch Events work together to address auditing and remediation requirements within a microservices architecture. All microservices are being tracked by CloudTrail and the audit trail is stored in an Amazon S3 bucket. CloudWatch Events sits on top of CloudTrail and triggers alerts when a specific change is made to your architecture.

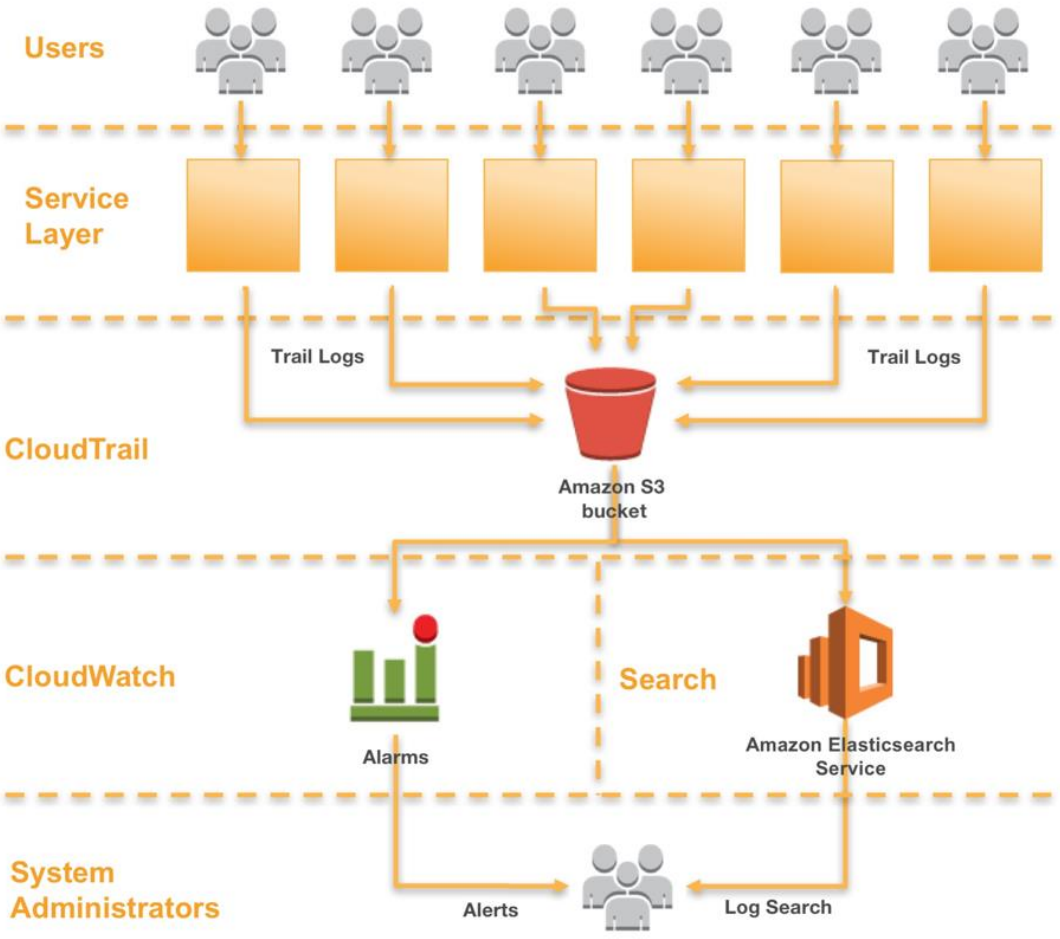

Figure 22: Auditing and remediation

Resource Inventory and Change Management

To maintain control over fast-changing infrastructure configurations in agile development teams, a more automated, managed approach to auditing and control of your architecture is beneficial.

AWS Config is a fully managed service that provides you with an AWS resource inventory, configuration history, and configuration change notifications to enable security and governance.[66] The AWS Config rules feature enables you to create rules that automatically check the configuration of AWS resources recorded by AWS Config.

While CloudTrail and CloudWatch Events track and respond to infrastructure changes across microservices, AWS Config rules allow a company to define security policies using specific rules and automatically detect, track, and alert violations to these policies.

In the example that follows, a developer team made a change to the API Gateway for his microservice to open up the endpoint to inbound HTTP traffic rather than allowing only HTTPS requests. Because this is a security compliance concern for the organization, an AWS Config rule is watching for this type of noncompliance, 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 creates an SNS notification.

Amazon SNS is used for two purposes in our scenario: 1) to send an email to a specified group to inform them about the security violation, and 2) to add a message into an SQS queue. The message is picked up from the SQS queue, and the compliant state is restored by changing the API Gateway configuration. This example demonstrates how it's possible to detect, inform, and automatically react to noncompliant configuration changes within your microservices architecture.

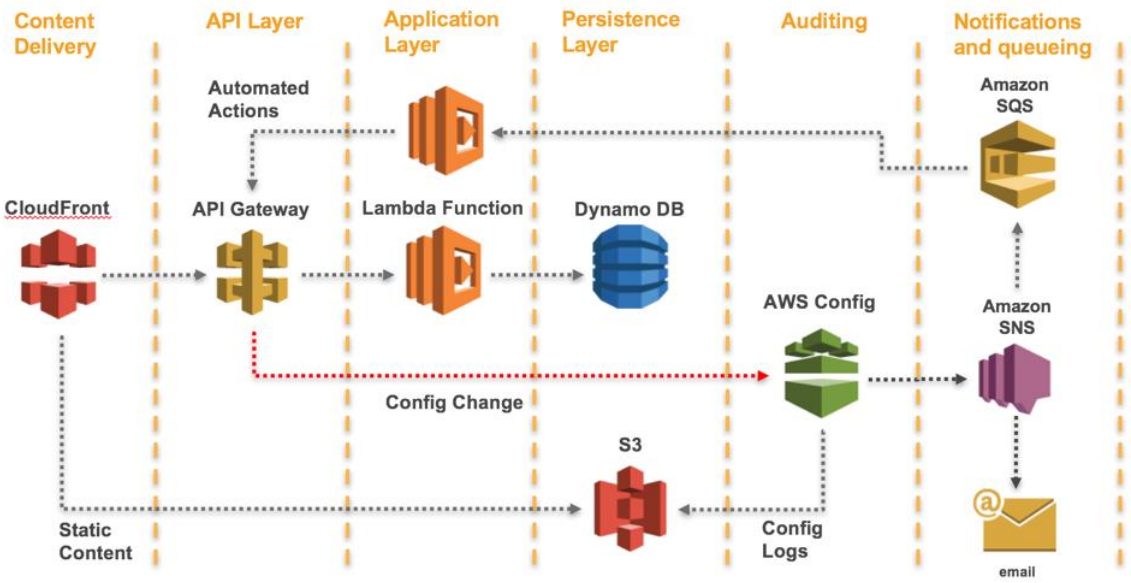

Figure 23: Detecting security violations with AWS Config

Conclusion

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

AWS offers a large portfolio of managed services that 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:

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

Document Revisions

Date	Description
September 2017	Integration of AWS Step Functions, AWS X-Ray, and ECS event streams.
December 2016	First publication

Notes

[1] https://www.google.com/trends/explore#q=Microservices

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

[3] https://en.wikipedia.org/wiki/DevOps

[4] https://en.wikipedia.org/wiki/Conway%27s_law

[5] http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

[6] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

[7] https://aws.amazon.com/devops/

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

[9] https://aws.amazon.com/devops/continuous-integration/

[10] https://aws.amazon.com/devops/continuous-delivery/

[11] https://aws.amazon.com/marketplace/b/2649367011

[12] https://aws.amazon.com/tools/#sdk

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

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

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

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

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

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

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

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

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

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

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

[24] http://swagger.io/

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

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

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

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

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

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

[31] http://docs.aws.amazon.com/elasticloadbalancing/latest/classic/using-domain-names-with-elb.html#dns-associate-custom-elb

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

[33] http://docs.aws.amazon.com/Route53/latest/DeveloperGuide/hosted-zones-private.html

[34] http://docs.aws.amazon.com/Route53/latest/DeveloperGuide/dns-failover-private-hosted-zones.html

[35] https://github.com/awslabs/ecs-refarch-service-discovery/

[36] https://aws.amazon.com/opsworks/

[37] https://github.com/Netflix/ribbon

[38] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html

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

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

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

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

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

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

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

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

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

[48] https://aws.amazon.com/kinesis/streams/

[49] https://aws.amazon.com/sqs/

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

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

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

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

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

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

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

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

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

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

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

[61] https://aws.amazon.com/kinesis/firehose/

[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/glacier/

[65] http://docs.aws.amazon.com/AmazonCloudWatch/latest/events/WhatIsCloudWatchEvents.html

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