Microservices on AWS

December 2016

amazon
web services

Notices

Contents

Introduction 1

Characteristics of Microservices 1

Benefits of Microservices 2

Challenges of Microservices 4

Microservices and the Cloud 6

Microservices on AWS 8

Simple Microservices Architecture on AWS 8

Reducing Operational Complexity 12

Distributed Systems Components 17

Conclusion 35

Contributors 36

Abstract

Microservices are an architectural and organizational approach to software development to speed up deployment cycles, foster innovation and ownership, and improve maintainability and scalability of software applications - as well as scaling organizations delivering 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

Characteristics of Microservices

The term *microservices* has received increased attention in the past few years.[1] Microservices architectures are not a completely new approach to software engineering, but rather a collection and combination of various successful and proven concepts such as object-oriented methodologies, agile software development, service-oriented architectures, API-first design, and Continuous Delivery.

Given that "microservices" is an umbrella term, it is hard to define it precisely. However, all microservices architectures share some common characteristics (see also Figure 1):

- **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. Those systems 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 around a set of capabilities and focuses on a specific domain. As soon as a component reaches a certain complexity, it might be a candidate to become its own microservice.
- **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 are usually heterogeneous with regard to operating systems, programming languages, data stores, and tools – an approach called polyglot persistence and programming.
- **Black box** – Individual microservice components are designed as black boxes, i.e., 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.[2] In addition to the benefit of allowing teams to progress independently at their own pace, 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 organizational aspect 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.[3]

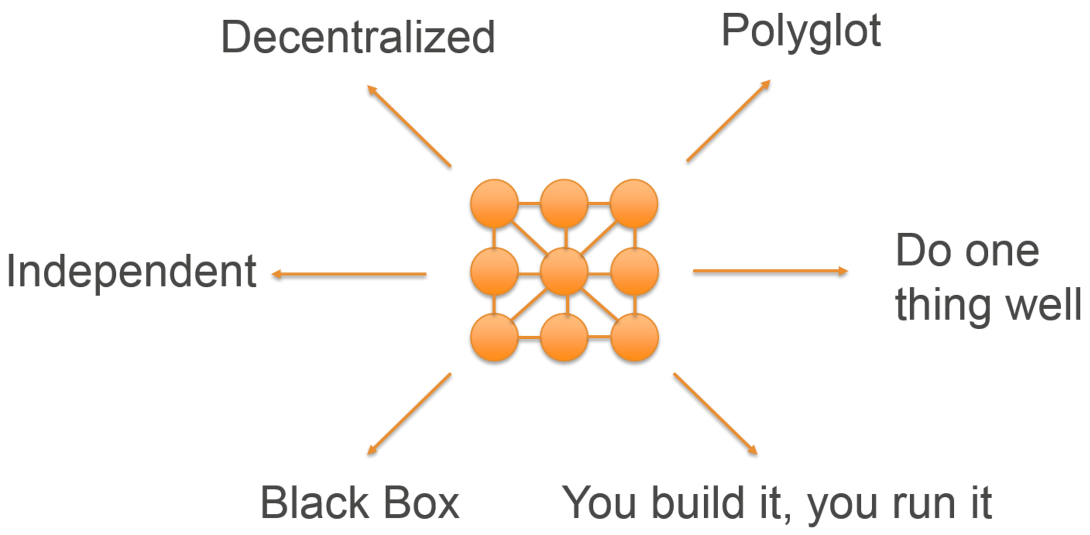

Figure 1: Characteristics of microservices

Benefits of Microservices

Many AWS customers adopt microservices to address limitations and challenges of traditional monoliths with regard to agility and scalability. 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 are empowered to work more independently and more quickly, thus shortening cycle times. You benefit significantly from the aggregate throughput of the organization.

Figure 2 illustrates the deployment process in an organizational structure with small independent teams as compared to a large team working on a monolithic deployment.

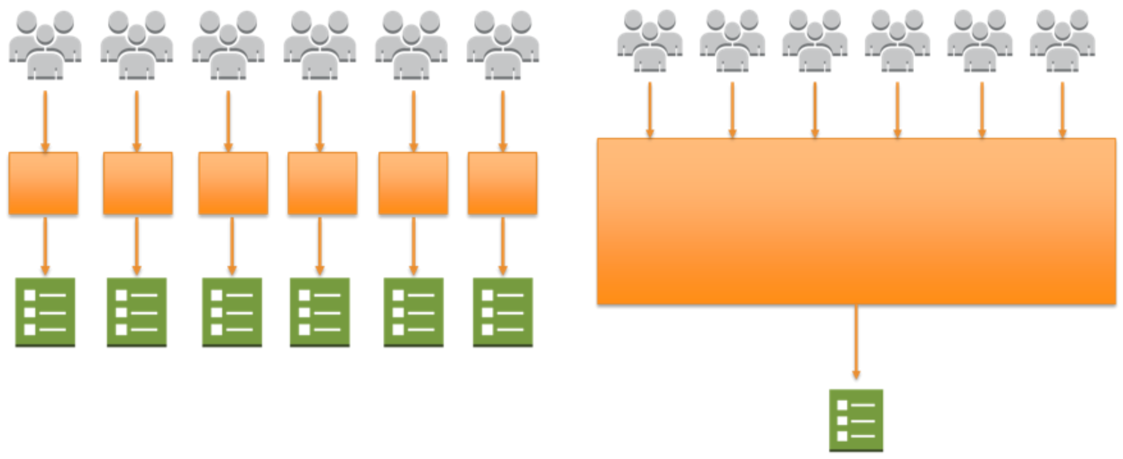

Figure 2: Deploying microservices

Innovation

The fact that small teams can act autonomously and choose the appropriate technologies, frameworks, and tools for their respective problem 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 releasing code—can be automated. It becomes easy to test new ideas 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, i.e., 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, i.e., 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 since failing components can be easily and automatically replaced.

Availability

Microservices architectures also make it easier to implement failure isolation. Techniques like 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 listed above, you should be aware that—as with all architectural styles—a microservices approach is not a silver bullet and brings some challenges and trade-offs.[4] For example:

- Microservices are effectively a distributed system, which brings a new set of problems often referred to as the *Fallacies of Distributed Computing*.[5]
- The migration process from a monolithic architecture to a microservices architecture requires you to determine the right boundaries for

microservices and to disentangle code dependencies going down to the database layer.

- Other challenges are of an organizational kind: 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 will mainly focus on the architectural and operational challenges of microservices. To learn more about DevOps and AWS, see https://aws.amazon.com/devops/.[6]

Architectural Complexity

While in monolithic architectures the complexity and the number of dependencies reside inside the code base, in microservices architectures complexity moves to the interactions of the individual services (Figure 3).

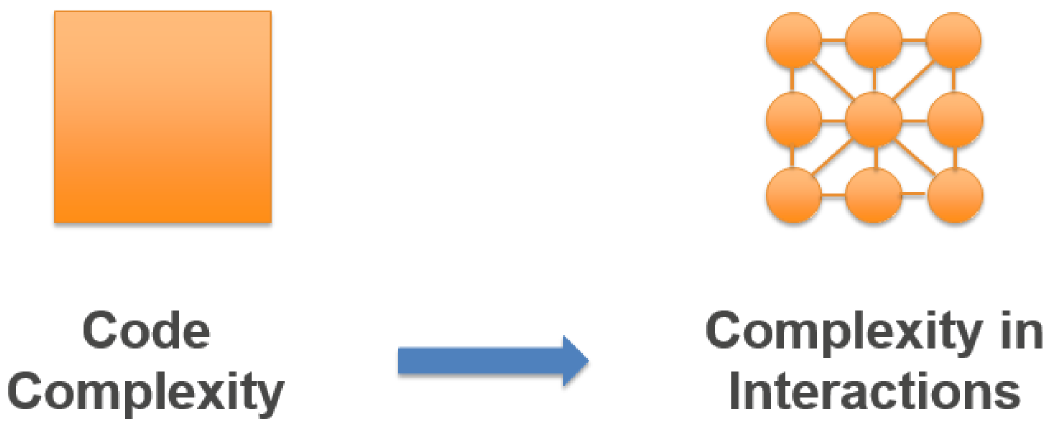

Figure 3: Complexity moves to interactions of individual microservices

Architectural challenges like dealing with asynchronous communication, cascading failures, data consistency problems, discovery, and authentication of services are topics to be addressed.

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 lack of standards and heterogeneous environments including different technologies and people skill sets?
- How to value diversity without locking into a multiplicity of different technologies that need to be maintained and upgraded over time?

Microservices and the Cloud

The popularity of microservices architectures is closely related to the rise of public cloud platforms such as AWS. This is not a coincidence. AWS has several characteristics that directly address the most important challenges of microservices architectures:

- **On-demand resources** – 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 naturally 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 in case they are not successful. Reducing cost and risk for experimenting with new ideas is a key element of driving innovation. This perfectly fits the goal of microservices to achieve high agility.

- **Programmability** –AWS services come with an API, Command Line Tools (CLI), and an SDK for different programming languages. Servers or even complete architectures can be programmatically cloned, shut down, scaled, monitored, and–in case of failure–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 for application code. As a consequence, any specific version of an infrastructure can be redeployed at any time. You can reason and get assurance about the quality and performance of a specific infrastructure version with regard to an application version. Rollbacks are no longer limited to the application—they also include the whole infrastructure.[7] Programmability and Infrastructure as Code are key elements to address operational complexity challenges related to provisioning, scaling, and maintaining microservices.

- **Continuous delivery** – The programmability of the cloud allows automation of the provisioning and deployment process. The idea of Continuous Integration within the development organization can be extended to the operations part of the application life cycle and enables Continuous Deployment and Delivery.[8, 9] Continuous delivery addresses operational complexity challenges related to managing 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 to reduce the operational complexity of running microservices architectures.

- **Service orientation** – AWS itself follows a microservice structure. Each AWS service focuses on solving a well-defined problem and communicates with other services via 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, runs many popular operating systems on Amazon Elastic Compute Cloud (Amazon EC2) that are available on the AWS Marketplace,[10] and supports a large variety of programming languages with SDKs.[11] This enables you to use the most appropriate solution for your specific problem without reinventing the wheel and duplicating processes.

Microservices on AWS

In this section, we first describe the four layers of a highly scalable, fault-tolerant microservices architecture (content delivery, API layer, application layer, and data persistence layer) and how to build it on AWS. We then recommend the AWS services that are best for implementing the different layers of a typical microservices architecture in order to reduce 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

Figure 4 depicts a reference architecture for a typical microservice on AWS. The architecture is organized along four layers: Content delivery, API layer, application layer, and persistence layer.

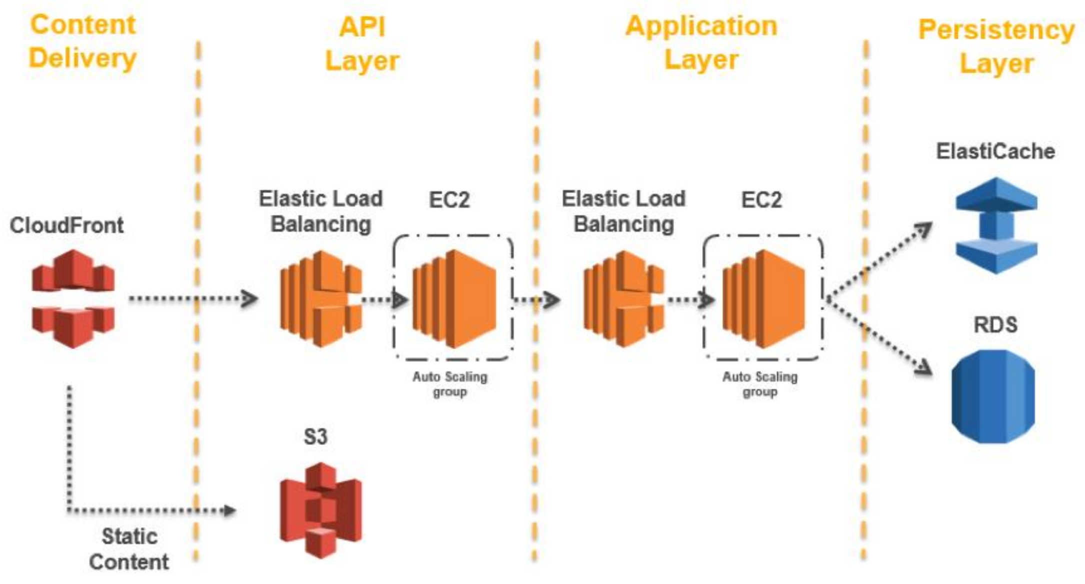

Figure 4: Typical microservice on AWS

Content Delivery

The purpose of the content delivery layer is to accelerate the delivery of static and dynamic content and potentially off-load the backend servers of the API layer.

Amazon CloudFront is a global content delivery network (CDN) service that accelerates delivery of your websites, APIs, video content or other web assets.[12]

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. Microservices running close to each other don't benefit from a CDN, but might implement other caching mechanisms to reduce chattiness and minimize latencies.

API Layer

The API layer is the central entry point for all client requests and hides the application logic behind a set of programmatic interfaces, typically an HTTP REST API.[13] The API layer accepts and processes calls from clients and might implement functionality such as traffic management, request filtering, routing, caching, or authentication and authorization.

Many AWS customers use Elastic Load Balancing (ELB) together with Amazon EC2 and Auto Scaling to implement an API layer.

Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances.[14]

The ELB load balancer distributes incoming requests to EC2 instances running the API logic.

Amazon EC2 is a web service that provides resizable compute capacity in the cloud.[15]

The EC2 instances are scaled out and 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.[16]

Application Layer

The application layer implements the actual application logic. Similar to the API layer, it can be implemented using Elastic Load Balancing, Auto Scaling, and Amazon EC2.

Persistence Layer

The persistence layer centralizes the functionality needed to make data persistent. Encapsulating this functionality in a separate layer helps to keep state out of the application layer and makes it easier to achieve horizontal scaling and fault-tolerance of the application layer.

Static content is typically stored on Amazon Simple Storage Service (S3) and delivered by Amazon CloudFront.

> Amazon S3 is object storage with a simple web service interface to store and retrieve any amount of data from anywhere on the web.[17]

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.[18] 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 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.

> Amazon Relational Database Service (RDS) makes it easy to set up, operate, and scale a relational database in the cloud.[19] It provides cost-efficient and resizable capacity while managing time-consuming database administration tasks, freeing you up to focus on applications and business.

Reducing Operational Complexity

The architecture we have described is already highly automated. Nevertheless, there's still room to further reduce the operational efforts needed to run, maintain, and monitor the different layers of a microservice component.

API Layer

Architecting, continuously improving, deploying, monitoring, and maintaining an API layer 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 (i.e., development, testing, and production) further multiply operational efforts.

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

Other important features and challenges include throttling requests to protect the backend, caching API responses, request and response transformation, and generating API definitions and documentation with tools such as Swagger.[20]

Amazon API Gateway addresses those challenges and reduces the operational complexity of the API layer.

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale.[21]

API Gateway allows you to create your APIs programmatically either by importing Swagger definitions or by using the AWS Management Console. API Gateway serves as a front door to any web application running on Amazon EC2, Amazon EC2 Container Service (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 Amazon CloudFront Point of Presence (PoP) to minimize latency and provide optimum user experience. API Gateway first checks if the request is in the cache and–if no cached records available–then forwards it to the backend for processing. Once the backend has processed the request, API call metrics are logged in Amazon CloudWatch and content is returned to the client.

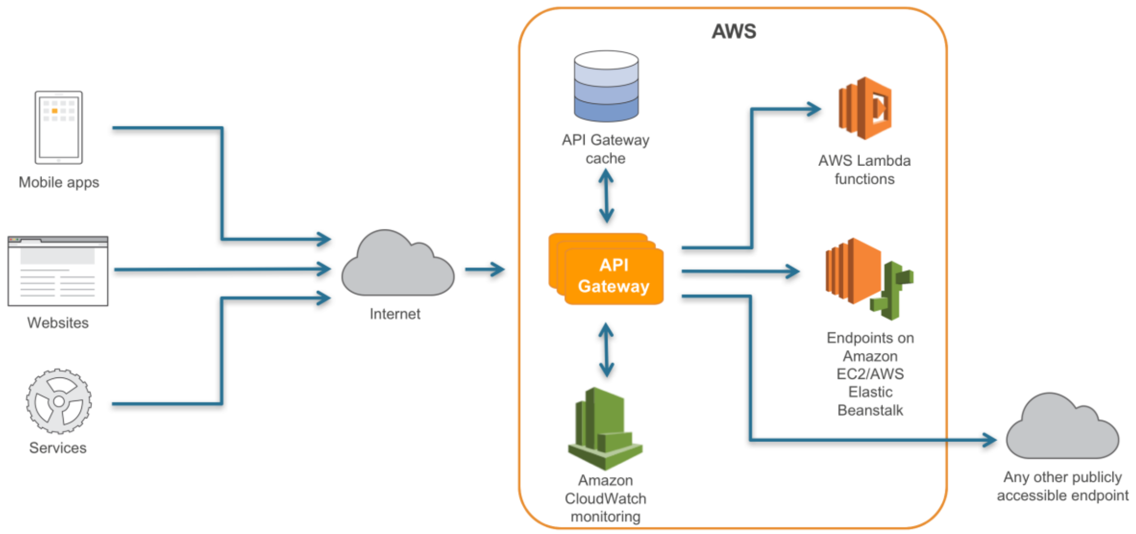
AWS
Mobile apps
Websites
Services
Internet
API Gateway cache
API Gateway
Amazon CloudWatch monitoring
AWS Lambda functions
Endpoints on Amazon EC2/AWS Elastic Beanstalk
Any other publicly accessible endpoint

Figure 5: API Gateway call flow

Application Layer

AWS provides several alternative options to running Elastic Load Balancing, Auto Scaling, or Amazon EC2. In this section, we look closer at those options and how they help to simplify deployment processes and possibly reduce operational complexity.

Reduce Management Complexity via Elastic Beanstalk

One option is to use AWS Elastic Beanstalk.

AWS Elastic Beanstalk is an easy-to-use service for deploying and scaling web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS.[22]

The main idea behind Elastic Beanstalk is that developers can easily upload their code and let Elastic Beanstalk automatically handle infrastructure provisioning and code deployment. Important infrastructure characteristics such as auto scaling, load balancing, and monitoring are part of the service.

Disposable Servers via Containers

Another approach to reduce operational efforts for deployment is container-based deployment. Container technologies like Docker[23] have increased in popularity in the last few years due to the following benefits:

- **Portability –** Container images are consistent and immutable, i.e., they behave the same no matter where they are run (on a developer's desktop as well as on a production environment).
- **Flexibility –** Containerization encourages decomposing applications into independent, fine-grained components, which makes it a perfect fit for microservices architectures.
- **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 OS.
- **Speed –** Containers are well-defined and reusable units of work with characteristics such as immutability, explicit versioning and easy rollback, fine granularity, and isolation. These characteristics help to significantly increase developer productivity and operational efficiency.

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.[24]

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, Elastic Load Balancing, Amazon Elastic Block Store (EBS) volumes, and AWS Identity and Access Management (IAM) roles.

Serverless Compute

“No server is easier to manage than no server”.[25] Getting rid of servers is the ultimate way to eliminate operational complexity.

AWS Lambda lets you run code without provisioning or managing servers.[26] 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 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 Amazon API Gateway. The possibility to make synchronous calls from Amazon API Gateway to AWS Lambda enables the creation of fully serverless applications and is described in detail in our documentation.[27]

Persistence Layer

While relational databases are still very popular and well understood, they 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.

> Amazon DynamoDB is a fast and flexible NoSQL database service for all applications that need consistent, single-digit millisecond latency at any scale.[28]

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. Amazon 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.

Putting it all together

Figure 6 shows the architecture of a serverless microservice where all the layers are built out of managed services, which eliminates the architectural burden to design for scale and high availability and eliminates the operational efforts of running and monitoring the microservice's underlying infrastructure.

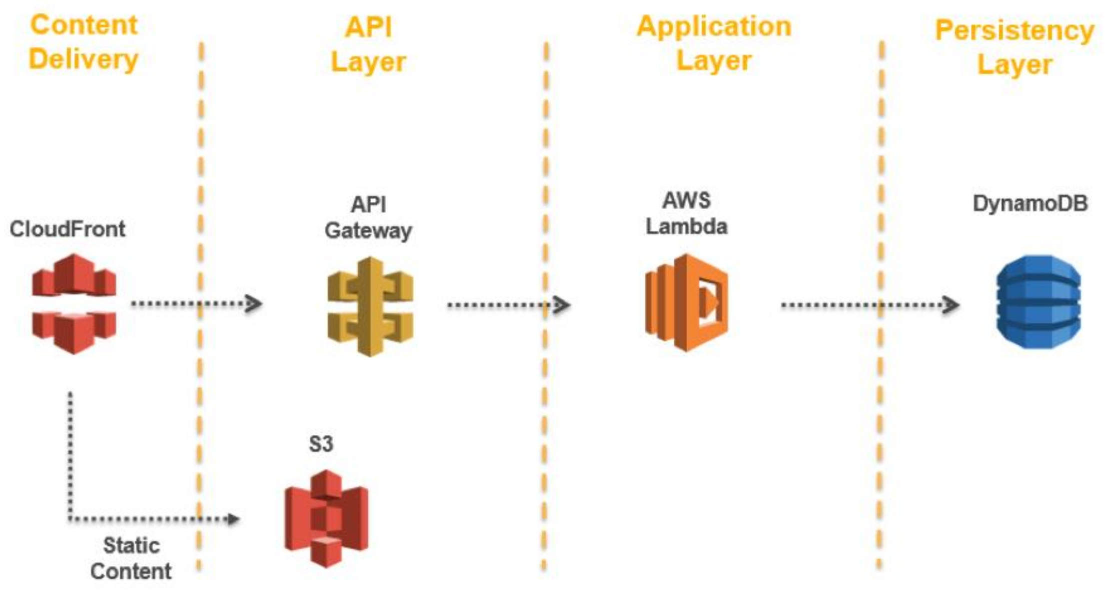

Figure 6: Serverless microservice

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 hard for services to communicate, but they also present interesting 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 is not 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.

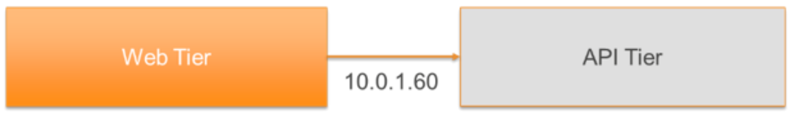

Figure 7: Client-side service discovery

Elastic Load Balancing-Based Service Discovery

One of the advantages of the Elastic Load Balancing service is that it provides health checks and automatic registration/de-registration of backend services in failure cases. Combining these features with DNS capabilities, it is 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 ELB load balancer's DNS name via a CNAME entry.[29] The DNS names of the service endpoints are then published across other applications that need access.

If you are using a container deployment model and leveraging Amazon ECS, have a look at our reference architecture that leverages CloudWatch Events and Lambda to register services into Amazon Route 53 private hosted zones in a fully automated way.[30]

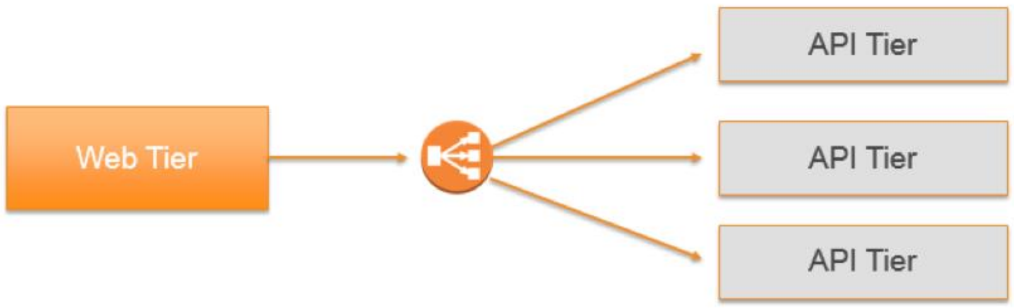

Figure 8: Elastic Load Balancing-based service discovery

DNS-Based Service Discovery

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

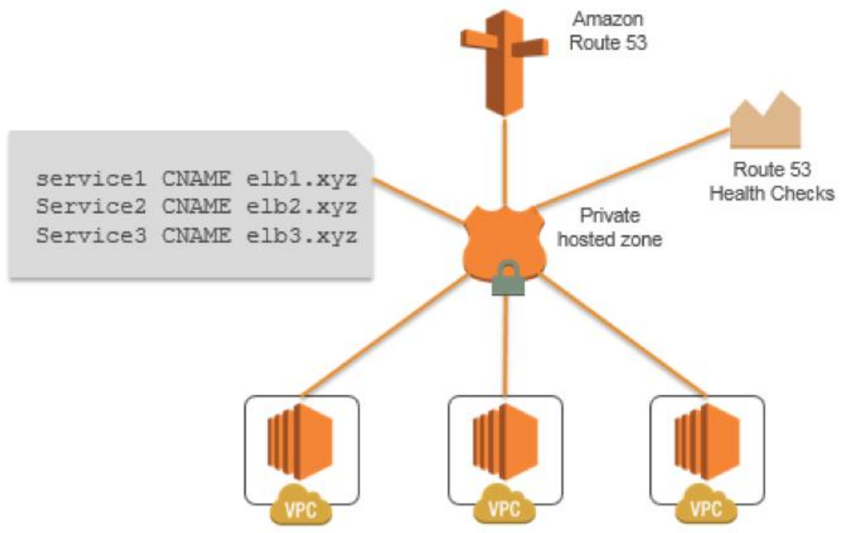

Figure 9: Domain Name System-based service discovery

Amazon Route 53 is a highly available and scalable cloud DNS web service.[31]

Amazon Route 53 provides several features that can be leveraged for service discovery. The private hosted zones feature allow it to hold DNS record sets for a domain or subdomains and restrict access to specific virtual private clouds (VPCs).[32] 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.[33]

Service Discovery Via Configuration Management

Using Configuration Management tools (like Chef, Puppet, or Ansible) is another means 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.

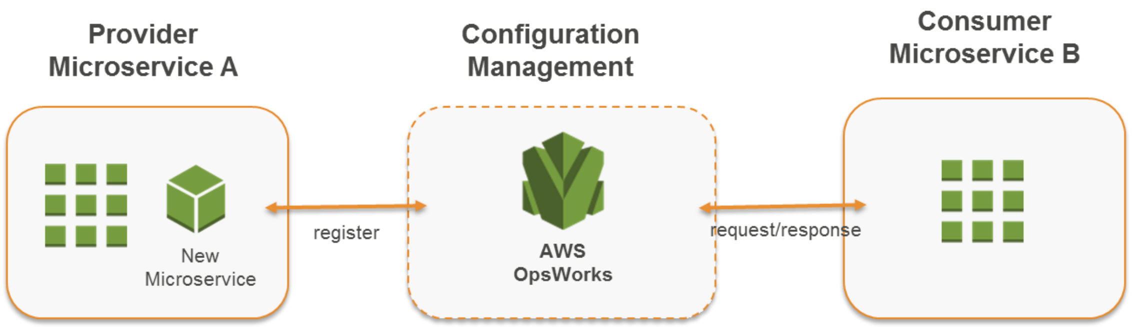

Figure 10: Service discovery using configuration management

Service Discovery Via 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 naturally works well with client-side load-balancing techniques such as Netflix Ribbon.34 Client-side load balancing can help eliminate bottlenecks and simplify management.

Figure 11 shows an architecture that leverages Amazon DynamoDB as a key-value store and Amazon DynamoDB Streams35 to propagate status changes to other microservices.

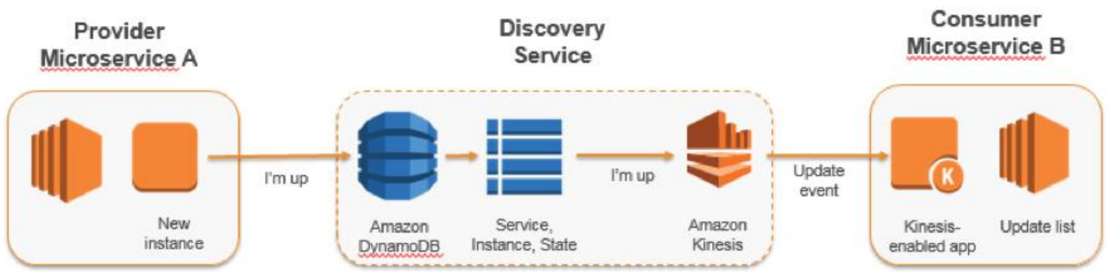

Figure 11: Service discovery using key/value store

Distributed Data Management

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

Distributed data management, however, raises new challenges. As a consequence of the CAP Theorem,[36] distributed microservices architectures inherently trade off consistency for performance and need to embrace eventual consistency.

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.[37] Using AWS Lambda with scheduled CloudWatch Events you can build a simple cleanup and deduplication mechanism.[38]

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.[39] 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.[40] In traditional data management systems, commands and queries are run against the same data repository.

Figure 12 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.

Amazon Kinesis Streams enables you to build custom applications that process or analyze streaming data for specialized needs.[41] Amazon 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 12 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 Amazon Kinesis Streams. All microservices run their own Amazon 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.

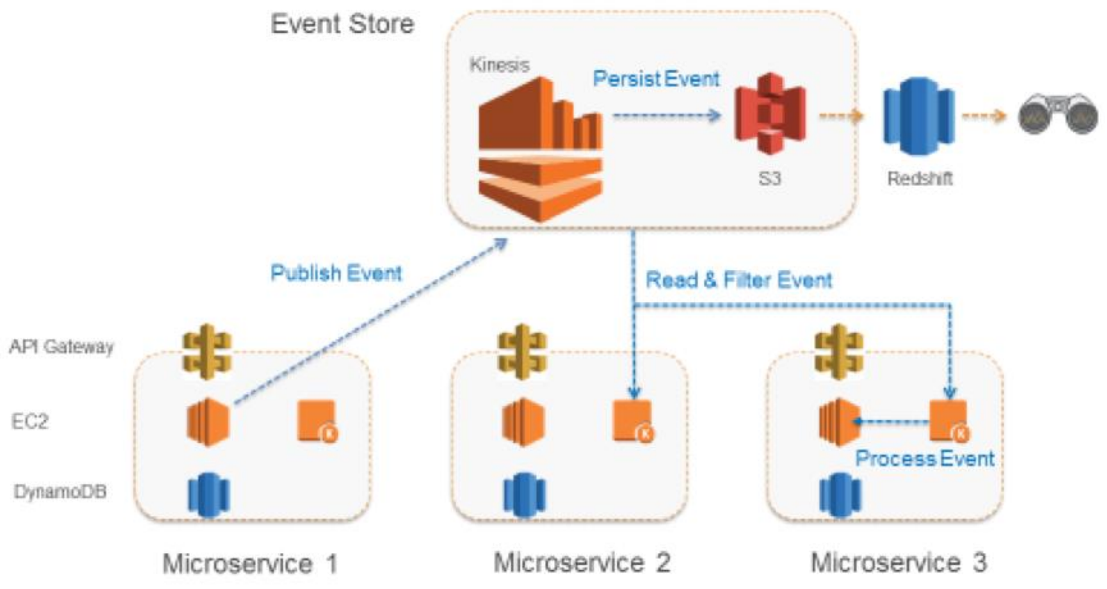

Figure 12: Event sourcing pattern on AWS

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. HTTP follows the Representational State Transfer (REST) architectural style, which relies on stateless communication, uniform interfaces, and standard methods.

Amazon API Gateway can be used to build, deploy, and manage RESTful API endpoints for backend services running on Amazon EC2, Amazon ECS, and AWS Lambda. An API object defined with the API Gateway service is a group of resources and methods. A resource is a typed object within the domain of an API and may have associated a data model or relationships to other resources. Each resource can be configured to respond to one or more methods, that is, standard HTTP verbs such as GET, POST, or PUT. REST APIs can be deployed to different stages and cloned to new versions.

Amazon 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.[42]

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.[43]

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 13 demonstrates the integration of Amazon SNS and Amazon SQS.

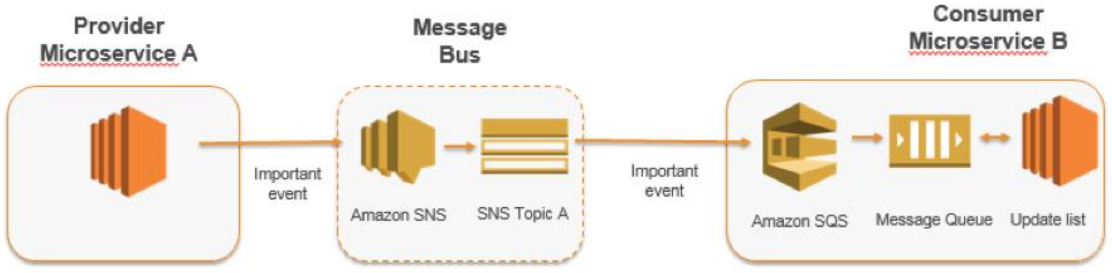

Figure 13: Message bus pattern on AWS

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.

Distributed Monitoring

A microservices architecture likely consists of many "moving parts" that have to be monitored.

Amazon CloudWatch is a monitoring service for AWS Cloud resources and the applications you run on AWS.[44]

You can use Amazon CloudWatch to collect and track metrics, centralize and monitor log files, set alarms, and automatically react to changes in your AWS environment. Amazon 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 Amazon CloudWatch to gain system-wide visibility into resource utilization, application performance, and operational health. Amazon 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 Amazon 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.[45]

Logging

Consistent logging is critical for troubleshooting and identifying issues. Microservices allow the production of 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.

Storing logs in one central place is essential to debug and get an aggregated view of distributed systems. In AWS you can use Amazon CloudWatch Logs to monitor, store, and access your log files from EC2 instances, AWS CloudTrail, or other sources. Amazon EC2 includes support for the `awslogs` log driver that allows the centralization of container logs to CloudWatch Logs.[46]

Centralizing Logs

You have different options for centralizing your log files. Most AWS services already centralize log files "out of the box." The primary destinations for log files on AWS are Amazon S3 and Amazon CloudWatch Logs. Figure 14 illustrates the logging capabilities of some of the services. Log files are a sensitive part of every system—almost every process on a system generates log files. A centralized logging solution aggregates all logs in a central location in order to be able to search and analyze these logs using tools like Amazon EMR, the elastic MapReduce service, or Amazon Redshift.

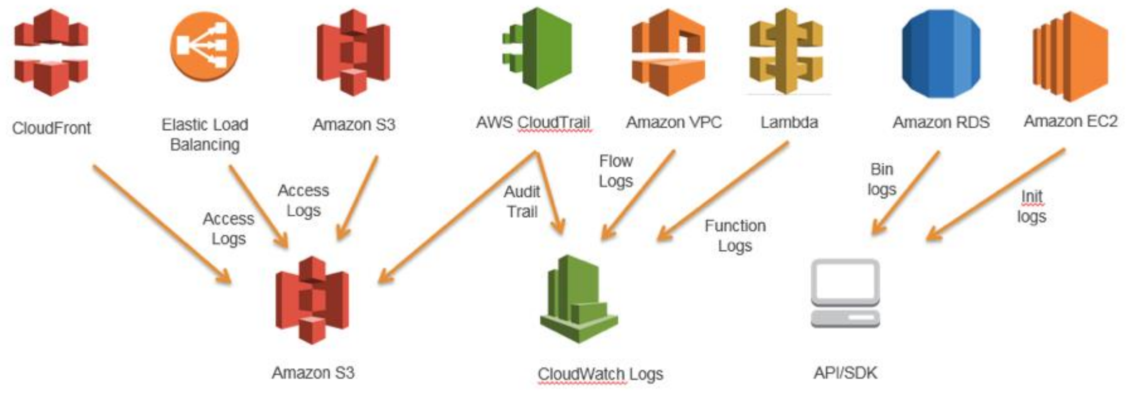

Figure 14: Logging capabilities of AWS services

Correlation IDs

In many cases, a set of microservices work 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.

The central idea of correlation ID is that a specific ID is created if a user-facing service receives a request. This ID can be passed along in the HTTP header (e.g., using a header field like "x-correlation-id") to every other service or in the payload that is passed. The ID can be included in every log message to find all relevant log messages for a specific request.

In order to get the correct order of calls in log files, it is a good approach to send a counter in the header that is incremented if the request flows through the architecture.

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 Elasticsearch Service (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.[47]

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 15 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 entries, etc. if anomalies, spikes, or other patterns of interest are detected in the data.

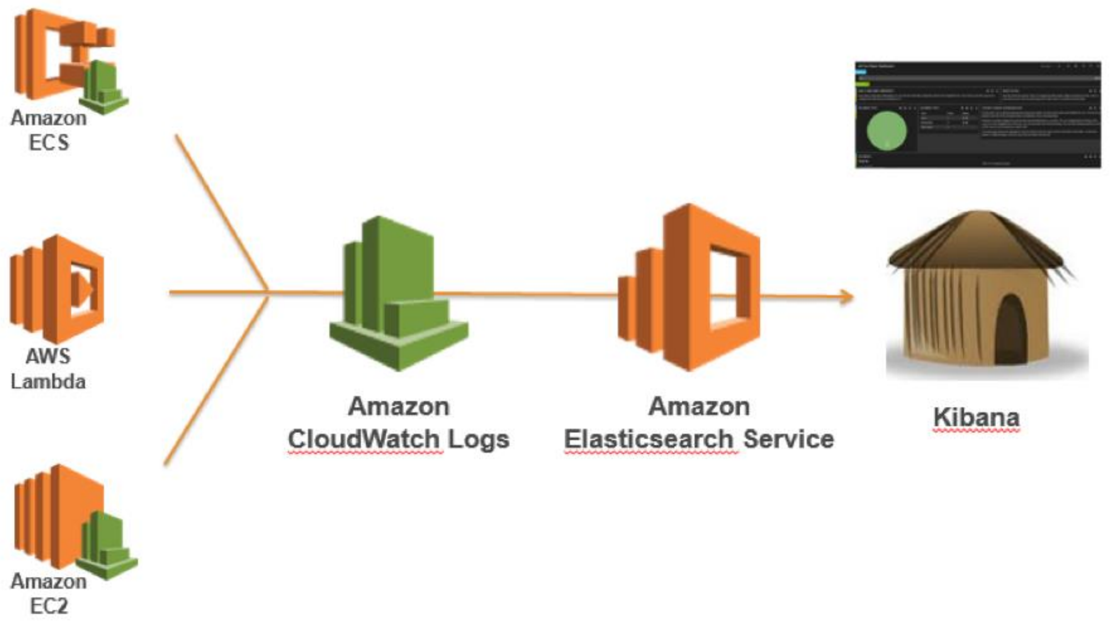

Figure 15: Log analysis with Amazon Elasticsearch Service and Kibana

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.[48]

Amazon QuickSight is a fast, cloud-powered business analytics service that makes to build visualizations, perform ad-hoc analysis, and quickly get business insights from your data.[49]

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

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

Amazon Kinesis Firehose is a fully managed service for delivering real-time streaming data to destinations such as Amazon S3, Amazon Redshift, or Amazon ES.[50]

Figure 16 depicts a scenario where log entries are streamed from different sources to Amazon Redshift using Amazon CloudWatch Logs and Amazon Kinesis Firehose. Amazon QuickSight uses the data stored in Amazon Redshift for analysis, reporting, and visualization.

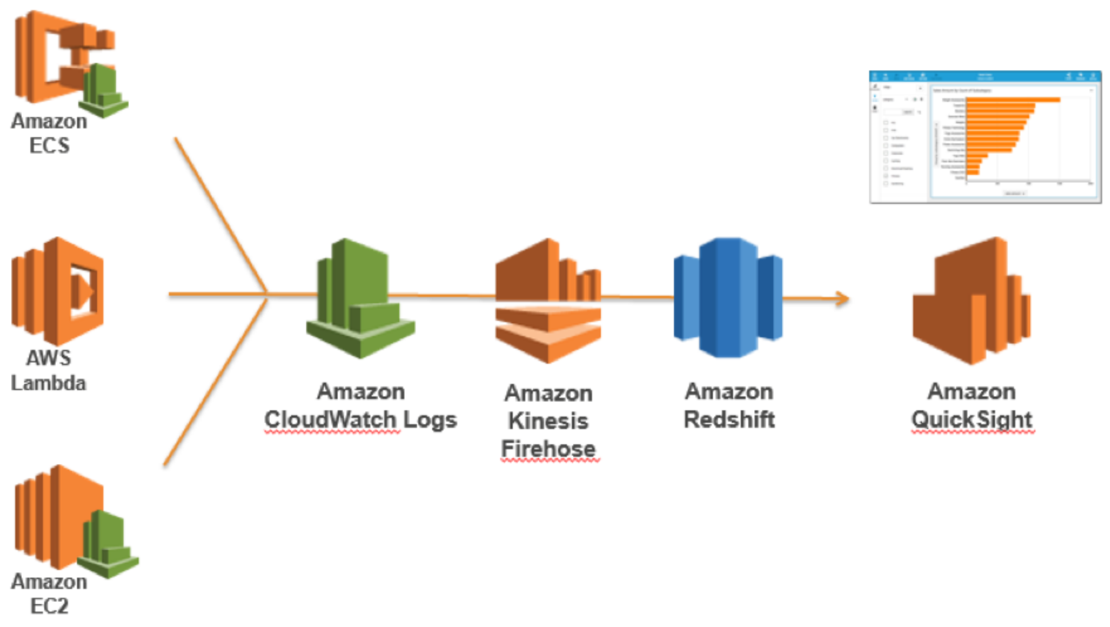

Figure 16: Log analysis with Amazon Redshift and Amazon QuickSight

Figure 17 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, e.g., Amazon Redshift or Amazon EMR, to analyze the data stored in the log stream and find anomalies.

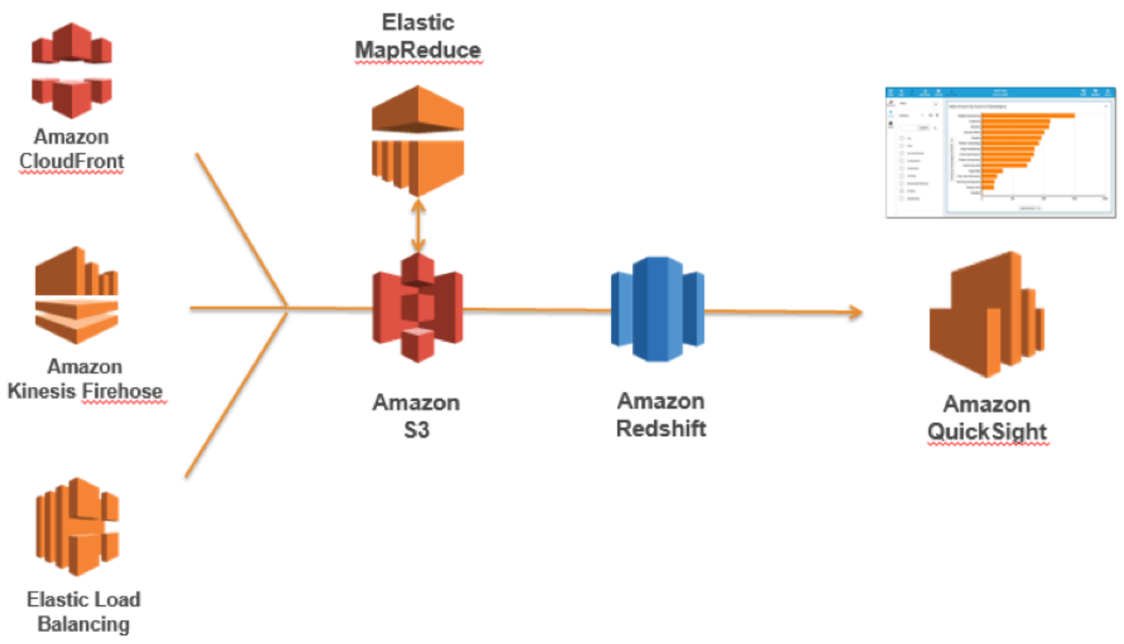

Figure 17: Log analysis on Amazon S3

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, e.g., 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 also useful to offload 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 platform to be logged and passed to either CloudWatch Logs in real time or to Amazon S3 within several minutes.

> AWS CloudTrail is a web service that records AWS API calls for your account and delivers log files to you.51 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.52

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.53 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 both log into 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.

Amazon CloudWatch Events delivers a near real-time stream of system events that describe changes in AWS resources.[54] Declarative rules associate events of interest with automated actions to be taken.

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

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

Figure 18 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 S3 bucket. CloudWatch Events sits on top of CloudTrail and triggers alerts when a specific change is made to your architecture.

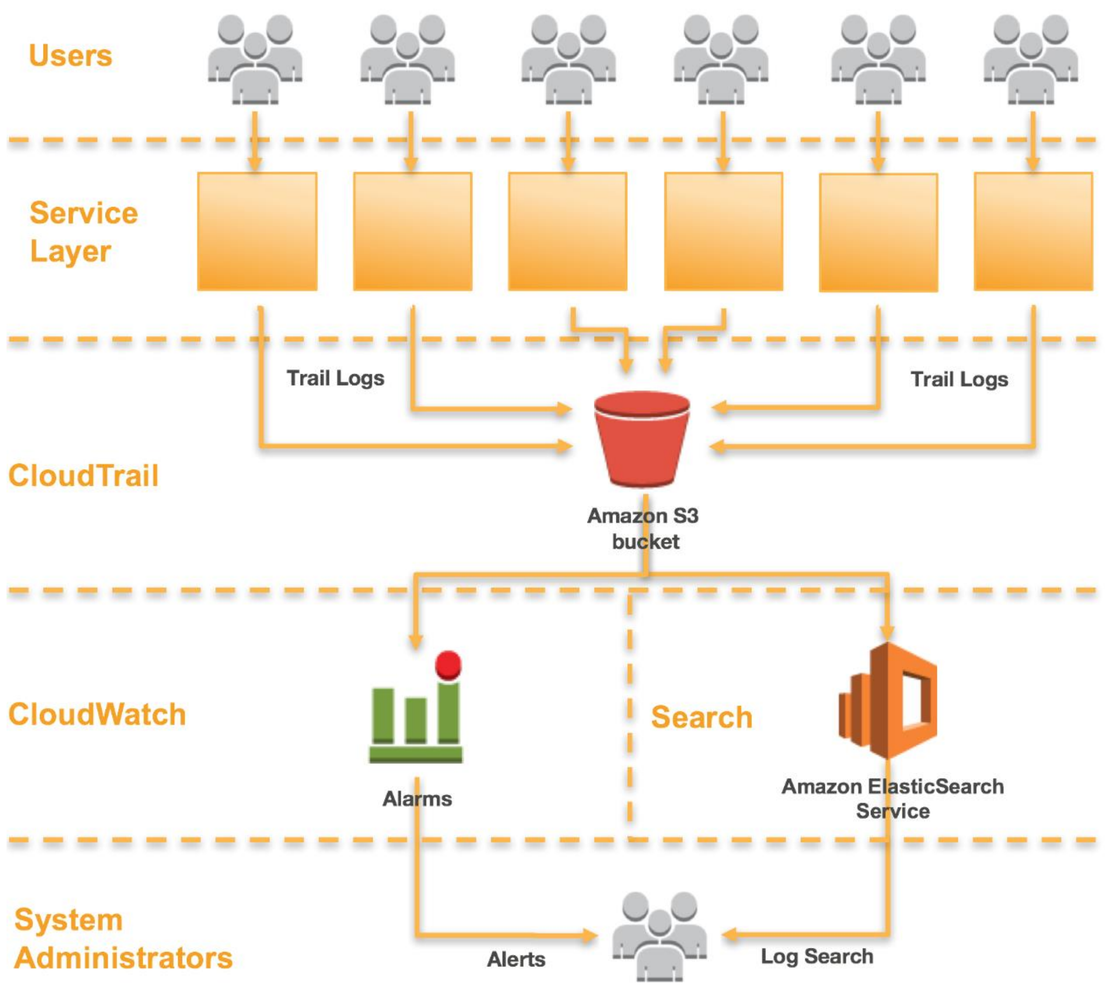

Figure 18: Auditing and remediation

Resource Inventory and Change Management

In order 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.[55] AWS Config Rules enables you to create rules that automatically check the configuration of AWS resources recorded by AWS Config.

While CloudTrail and CloudWatch Events are important building blocks to track and respond to infrastructure changes across microservices, AWS Config Rules allows a company to define security policies via specific rules and automatically detect, track, and alert violations to this policy.

In the example that follows, a member of a development team has made a change to the API Gateway for his microservice to open up the endpoint to inbound HTTP traffic rather than only allowing HTTPS requests. Because this is a security compliance concern for the example organization, an AWS Config Rule is watching for this, 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 about the security violation, and 2) to add a message into an SQS queue. From there, the message is picked up, and the compliant state is restored by changing the API Gateway configuration. This example demonstrates how it is possible to detect, inform, and automatically react to non-compliant configuration changes within your microservices architecture.

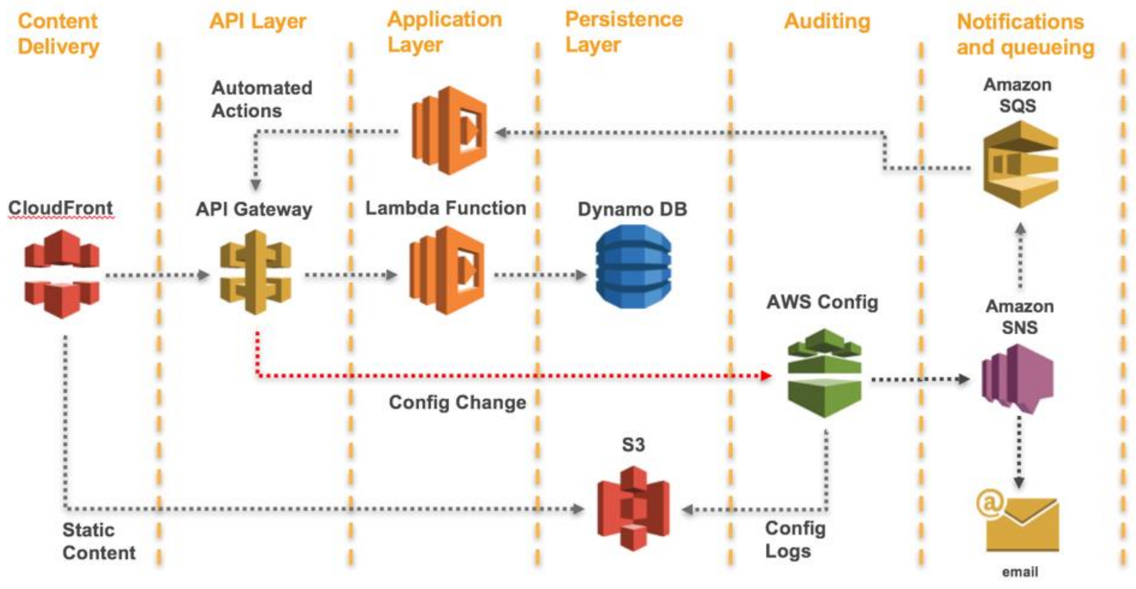

Figure 19: Detecting security violations with AWS Config

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 a couple of challenges that may 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

Notes

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

2 https://en.wikipedia.org/wiki/DevOps

3 https://en.wikipedia.org/wiki/Conway%27s_law

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

5 https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

6 https://aws.amazon.com/devops/

7 https://aws.amazon.com/cloudformation/

8 https://aws.amazon.com/devops/continuous-integration/

9 https://aws.amazon.com/devops/continuous-delivery/

10 https://aws.amazon.com/marketplace/b/2649367011

11 https://aws.amazon.com/tools/#sdk

12 https://aws.amazon.com/cloudfront/

13 https://en.wikipedia.org/wiki/Representational_state_transfer

14 https://aws.amazon.com/elasticloadbalancing/

15 https://aws.amazon.com/ec2/

16 https://aws.amazon.com/autoscaling/

17 https://aws.amazon.com/s3/

18 https://aws.amazon.com/elasticache/

19 https://aws.amazon.com/rds/

20 http://swagger.io/

21 https://aws.amazon.com/api-gateway/

22 https://aws.amazon.com/de/elasticbeanstalk/

23 https://www.docker.com/

24 https://aws.amazon.com/ecs/

25 https://twitter.com/awsreinvent/status/652159288949866496

26 https://aws.amazon.com/lambda/

27 http://docs.aws.amazon.com/apigateway/latest/developerguide/getting-started.html

28 https://aws.amazon.com/dynamodb/

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

30 https://github.com/awslabs/ecs-refarch-service-discovery/

31 https://aws.amazon.com/route53/

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

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

34 https://github.com/Netflix/ribbon

35 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html

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

37 https://en.wikipedia.org/wiki/Master_data_management

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

39 http://martinfowler.com/eaaDev/EventSourcing.html

40 http://martinfowler.com/bliki/CQRS.html

41 https://aws.amazon.com/kinesis/streams/

42 https://aws.amazon.com/sqs/

43 https://aws.amazon.com/sns/

44 https://aws.amazon.com/cloudwatch/

45 https://docs.aws.amazon.com/autoscaling/latest/userguide/policy_creating.html

46 https://docs.aws.amazon.com/AmazonECS/latest/developerguide/using_awslogs.html

47 https://aws.amazon.com/elasticsearch-service/

48 https://aws.amazon.com/redshift/

49 https://aws.amazon.com/quicksight/

50 https://aws.amazon.com/kinesis/firehose/

51 https://aws.amazon.com/cloudtrail/

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

53 https://aws.amazon.com/glacier/

54 http://docs.aws.amazon.com/AmazonCloudWatch/latest/events/WhatIsCloudWatchEvents.html

55 https://aws.amazon.com/config/