Dung Nguyen

SERVERLESS ARCHITECTURE ON AWS

SERVERLESS ARCHITECTURE ON AWS

Dung Nguyen
Bachelor’s Thesis
Autumn 2019
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme of Information Technology

Author: Dung Nguyen
Title of the bachelor’s thesis: Serverless Architecture on AWS
Number of pages: 51
Supervisor: Kari Laitinen
Term and year of completion: Autumn 2019

This thesis aimed at extending knowledge on the serverless architecture and its possibility on Amazon Web Services. Additionally, under the scope of this thesis, a comparison between different cloud providers was also conducted to strengthen the fact that AWS would be the solid choice for serverless development.

In general, the serverless architecture can help organizations to reduce operational costs while maintaining a high-performance system which scales based on the load. AWS has a wide range of services which can support the serverless development and deployment at the advanced level. The thesis is divided into three main parts: Part I focuses on theories and fundamentals of the serverless architecture, Part II contains a comparison of different cloud providers and in-depth research on AWS serverless platform. Finally, Part III provides the feasibility of developing a serverless application on AWS through a practical example.

The result of this thesis can be promoted as a reference for the serverless architecture and why it should be taken into consideration by organizations. Besides, choosing the right cloud provider from the beginning is also crucial and AWS was proved to be a stable contender in the serverless era.

Keywords: AWS, serverless, system architecture

PREFACE

This thesis was conducted as the final project of my study at Oulu University of Applied Sciences. The aim of the project was to strengthen my knowledge on the serverless architecture which can boost my contribution to my daily work at Haltian. I would like to express my gratitude towards my company – Haltian and my supervisor Mr. Kari Latinen. He has supported the idea from the beginning and guide me to come up with the proper outline. I would also like to thank Mrs. Kaija Posio for reviewing my writing in this thesis.

Besides, I would like to thank my girlfriend, who has always been by my side and supported me for this thesis work.

Oulu, 04.08.2019
Dung Nguyen

CONTENTS

VOCABULARY

API: Application Programming Interface

AWS: Amazon Web Services

BaaS: Backend as a Service

CLI: Command Line Interface

CPU: Central Processing Unit

EC2: Elastic Compute Cloud

FaaS: Function as a Service

GCP: Google Cloud Platform

HTTP: Hypertext Transfer Protocol

IaaS: Infrastructure as a Service

IT: Information Technology

PaaS: Platform as a Service

RAM: Random Access Memory

SOA: Service Oriented Architecture

SaaS: Software as a Service

S3: Simple Storage Service

SNS: Simple Notification Service

SOA: Service Oriented Architecture

SQS: Simple Queue Service

URL: Uniform Resource Locator

VM: Virtual Machine

VPS: Virtual Private Server

1 INTRODUCTION

Technologies evolve from time to time, which opens up a variety of new possibilities to build extraordinary things that we have never thought of before. As a result, the industrial standard of software has gone up, and so do the requirements from users. Businesses now strive to focus on improving user experiences as well as competing with other competitors. With that being said, time to market, which is the needed time for businesses to go from a business idea to a real product, is very crucial in this era. To materialize a business idea, there are various ways of utilizing technologies to design a system from ground-up which suits the business model. Choosing the right technology while ensuring the scalability of the system as well as keeping the cost at a minimum is a challenging problem and should be taken into consideration.

Serverless technologies were invented as a solution to the problem that many organizations are facing in today's market. By taking serverless technologies into use, organizations now have a way to eliminate idle, underutilized servers to reduce the costs. Indeed, analysts estimate that around 85% of servers in practice have underutilized capacity, which is proved to be costly and wasteful [1]. In addition to saving costs, organizations now also have the ability to build the microservices, event-driven systems that can scale automatically depending on the load, thus improving time to market.

This thesis was conducted to provide an in-depth study about the serverless architecture and how to apply it in real-world contexts. All cloud providers are now jumping to the serverless war since they know it is going to be the future of the system architecture. Choosing the right cloud provider is also under the scope of this thesis since it is not straightforward to change to another provider after committing to one. Needless to say, having a comprehensive knowledge about this architecture and understanding when to apply it will bring substantial advantages for companies towards their competitors.

2 SERVERLESS FUNDAMENTALS

Serverless, undoubtedly, is currently one of the most popular topics in the software architecture world. Before diving into the definition of the term “serverless”, the evolution of system architecture must be introduced first to see why there is such a change which leads to the current state of the architecture today. Then, the discussion on the advantages and disadvantages of the serverless architecture would act as a pre-cursor for companies when reasoning whether it would be a good fit for them.

2.1 The evolution of system architecture

Pre-virtualization era

This is the beginning of the system architecture. In this era, organizations need to set up a physical server or a fleet of servers called a rack to deploy their software system (on-premises model). Components inside a physical server are depicted in the Figure 1. The Figure 2 demonstrates how server racks in a data center look like in reality. Those servers are usually put inside a data center with high-speed networks and excessive power supplies to ensure that the system is running 24/7. In the data center, there are normally a group of infrastructure engineers who work together to install the servers on the rack, power them up and connect them to the network. After that, an operating system needs to be installed to those servers together with various software dependencies such as web servers, databases, and caches needed for the system. Then, the system source code has to be copied to each server. Besides, servers need to be constantly monitored, and patches need to be applied regularly to prevent any vulnerabilities [3]. Needless to say, the process is repetitive and has to be performed again and again whenever there are new servers.

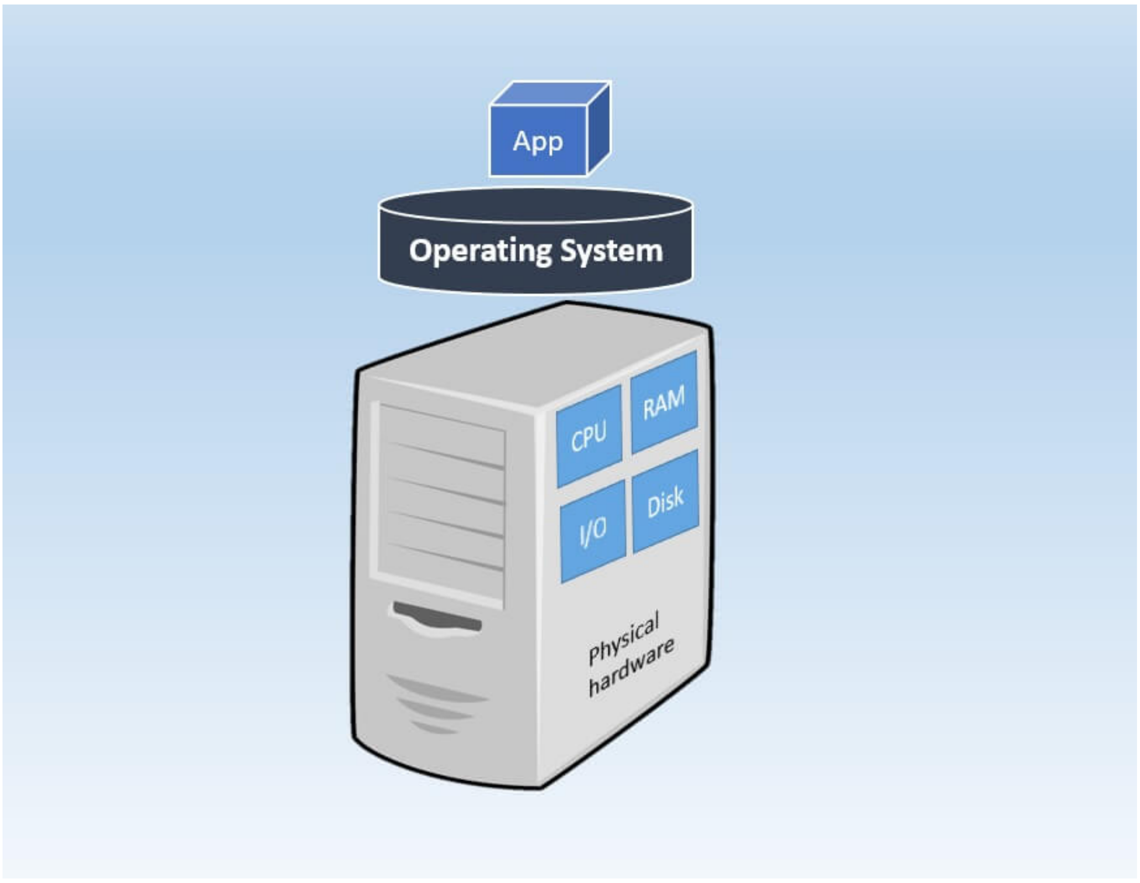


FIGURE 1. Components of a physical server [4]

FIGURE 2. Server racks in a data center [5]

With this kind of system architecture, companies need to spend a substantial amount of money on human resources just to maintain the system. The maintenance cost could exceed the development cost in the long term. Additionally, the system cost is also increased unnecessarily due to the fact that there have to be more servers than needed to ensure the availability during the peak load time, which leads to various under-utilized servers.

During this time, there is a service on the market called “bare metal servers rental” which allows companies to rent the physical servers. This service model is called IaaS, which stands for an Infrastructure as a Service. Service providers will take care of the maintenance process. Using the service will help organizations reduce the maintenance cost as well as infrastructure cost to some extent.

Virtualization era

The next advancement in the system architecture world is made possible by the virtualization technology. By utilizing software called “hypervisor” which mimics a physical server, one physical server can now run multiple virtual servers with a different operating system for each server, as shown in the Figure 3 [3].

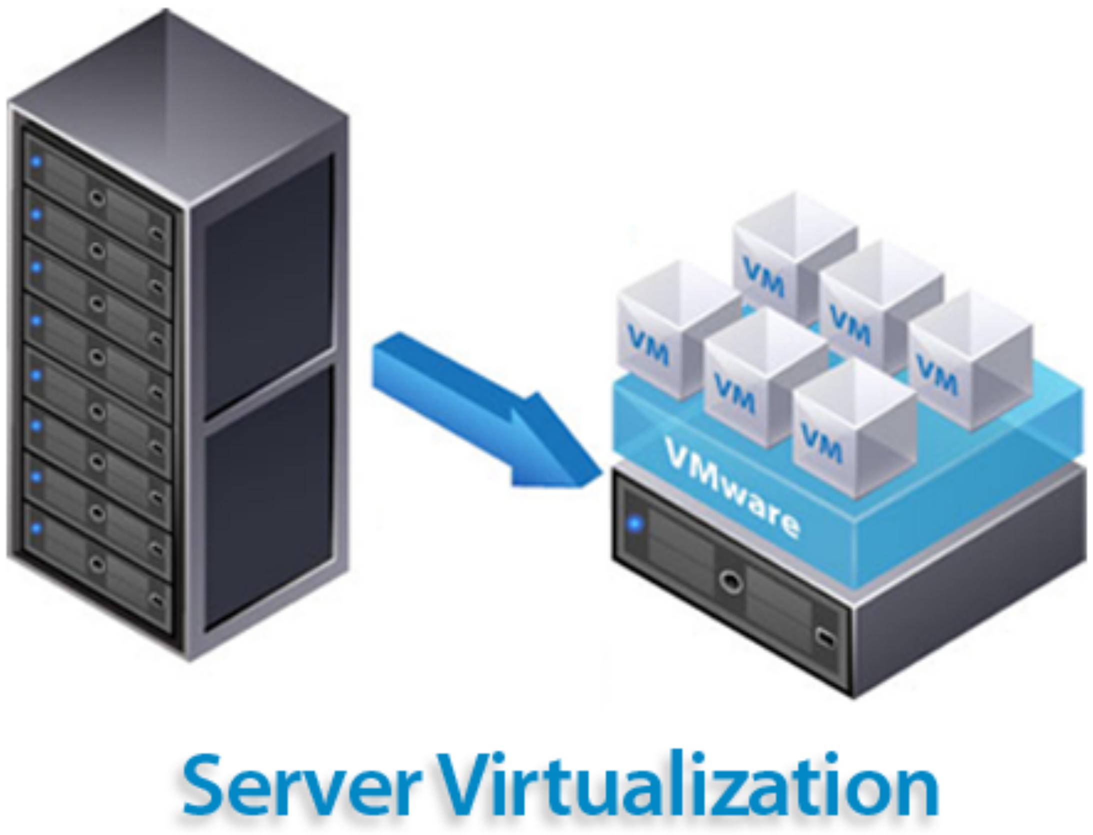


FIGURE 3. The transition of physical servers to virtual servers [6]

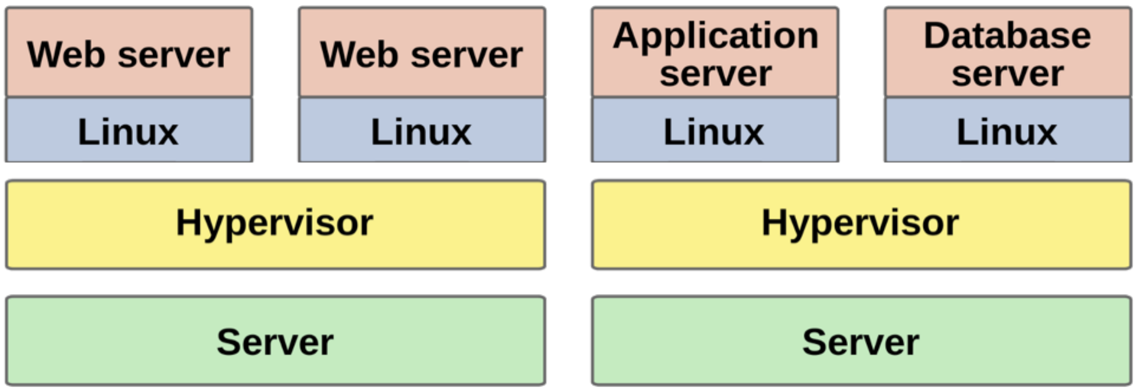


FIGURE 4. The virtualization server architecture [3]

As can be seen in the Figure 4, there are two physical servers, in which there are four different virtual servers with the support of the hypervisor software. There are two web servers running in the first physical server, one application server and one database server running in the second physical server, all using Linux as their operating system. The most popular application to run this kind of virtualization is called VMware Workstation.

Even though they are called "virtual" servers, each server instance is completely isolated from others, meaning that they use separate RAM and CPUs which are assigned to them from the physical server. Moreover, a failure in one server does not cascade to others. Another benefit of using virtualization is that engineers can create a "snapshot" of the current state of the server [3]. Thus, the snapshot can be transferred to other servers to create many replications of the original server.

The existence of virtualization opens up a new market in the IT industry which is called a VPS hosting service. In this service, the providers set up server racks in which the virtualization software is installed. After that, one physical server will host multiple virtual servers in different sizes which then can be rented out by users. By using the service, developers will have some benefits, such as:

1. The server will only need to be set up one time. Then a snapshot will be created which can be replicated to as many servers as is necessary. The number of instances can scale up or down on demand.
2. If there is a failure in the underlying physical server, the system can be recovered by putting the snapshot to another physical server.
3. Developers can choose different types of server instances depending on the functionality and the load of the component.

Containerization era

After the wave of virtualization technology, people seem to be not fully satisfied with the advantages brought to them. A new challenge has been raised which is how to reduce the deployment time as well as the size of the deployment package. Virtual machines have slow deployment time because they have to boot up the whole operating system. In addition, since it requires an operating system to run, the deployment package needs to include that as well which leads to an increase in size. Thus, a container technology was born as an answer to the challenge. In 2007, a contribution to the Linux kernel made by Google brought the possibility to perform the virtualization in a limited form without the need of the hypervisor layer, which is called “containers” [3].

The term “containers” comes from the physical containers which are used to store various objects for transportation purposes (e.g. a container ship). By using that kind of physical containers, a standard way to carry goods around is established. Different means of transportation only need to ensure that it has enough space for a container without concerning how to sort different kinds of goods inside the container [7]. Having a similar idea, a container in the software world can pack an application with all of its dependencies into something called a container image. Container images can be placed into different machines in which they will span running containers utilizing the host operating system and containing the underlying applications.

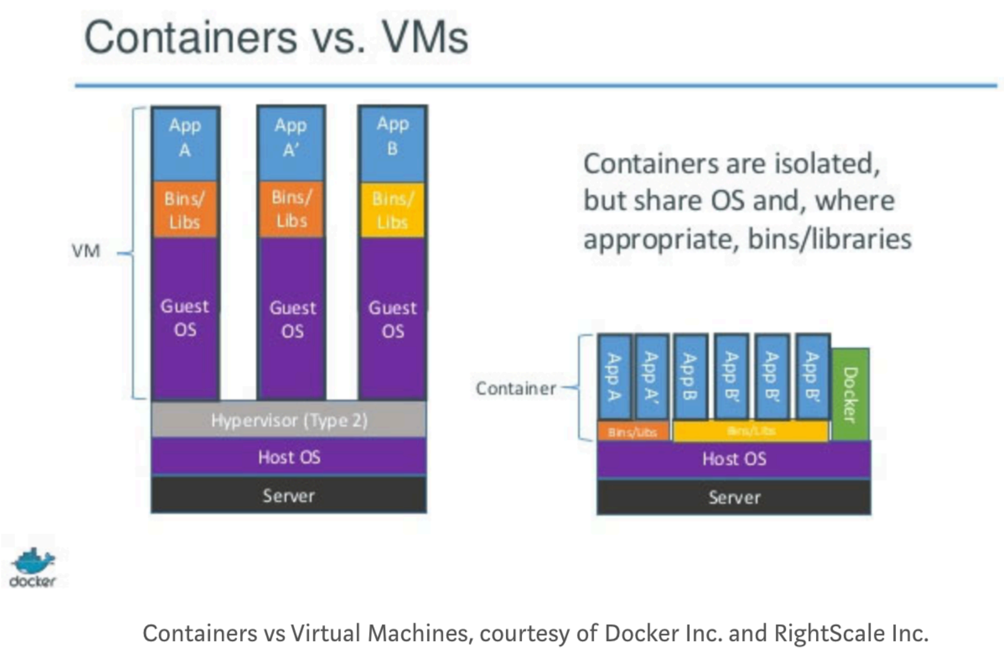


FIGURE 5. Containers vs VMs [3]

The Figure 5 demonstrates the differences between containers and VMs. There are many reasons why organizations might favor containers over virtual machines. Firstly, the container technology brings the possibility of replicating exactly the production environment to different developments such as local, test, and staging. In that way, developers and testers can ensure what works in the local development and the staging environment will work on the production environment. Moreover, it also eliminates the problem of having different versions of dependencies in different environments. Secondly, a single server can host many more containers than virtual machines due to the difference in size (tens of megabytes vs several gigabytes). Thirdly, the deployment time will be vastly reduced for containers since they do not require to boot up an entire operating system like virtual machines. Thus, containers can be spanned and discarded on demand. Finally, containers leverage the idea of “microservices”, where each component in the application can be a container or a set of containers called a module. Each team inside the company can take over one or more isolated modules and develop them in parallel with other teams which increases the development as well as the maintenance time of the application [8].

Serverless era

The first mark of the serverless era is the introduction of PaaS (Platform as a Service). The reason why PaaS exists is that providers observe that developers tend to create the same type of application over and over. They use a certain set of programming languages together with some popular frameworks and dependencies (e.g. database solutions, proxy servers) which leads to the fact that providers can just manage those components for them and what is left for them is the business logic of their application. Management tasks for providers include setting up virtual or physical servers, and installing an operating system with software dependencies, such as language runtime, and database engines [3]. This type of service can be considered as “serverless” since developers do not need to concern about servers anymore. Some popular PaaS services are: Heroku, AWS Elastic Beanstalk, Google App Engine.

As time went by, service providers realized that applications usually run on request, rather than staying online all the time. By only initializing servers and executing the business logic when there is a request, providers can distribute the requests across the set of servers, thus utilizing the resource more efficiently [3]. Moreover, providers can also support scalability seamlessly since each request is stateless and short-lived, meaning that a request can be served by any of the servers. For instance, if there are 500 concurrent requests, there will be 500 servers which are spanned immediately to serve all the requests. After executing the requests, those servers will be turned off and ready to be initialized again when there is a new request. This model is called FaaS (Functions as a Service) and it is the key component in the event-driven architecture where the logic is only executed when there are events happening in the system (e.g. an API request, a database update or an image uploaded).

From the developer's perspective, this model brings various perks over the traditional one. Firstly, they only need to focus on the core business logic code, meaning that the code will be much shorter and focus only on serving a single client. As a result, writing the code for FaaS is the same as writing the code for a function in a normal program where the logic is to transform a set of inputs to an output. The whole program is built by connecting many FaaS functions together. Secondly, they do not need to take provisioning into consideration, it should be the job of the providers. The development environment will be the same as the production development since the system will scale up or scale down automatically according to the load. Finally, as said above, they only need to pay for the execution time which is spent on requests, there is nothing called "idle time" in this FaaS mode [3].

With the existence of FaaS, providers even pushed it further by not only providing functions as a service but also various serverless services. Some popular services currently are serverless databases, serverless file storages, and serverless message queues [3]. Those services have several common characteristics. First, the same as serverless functions, there is no provisioning requirement. Users do not need to specify how many instances the service should have. The capacity of the service will automatically increase if needed.

Second, users only need to pay for the amount of usage. A serverless file storage system is a case in point. In this type of service, users usually only have to pay for the total size of all the files (e.g. 50GB) that they store in the service. There is mostly no limit on the storage size, the service will scale on demand.

In summary, the main difference between different service models is the level of abstraction in each service, which is shown in the Figure 6.

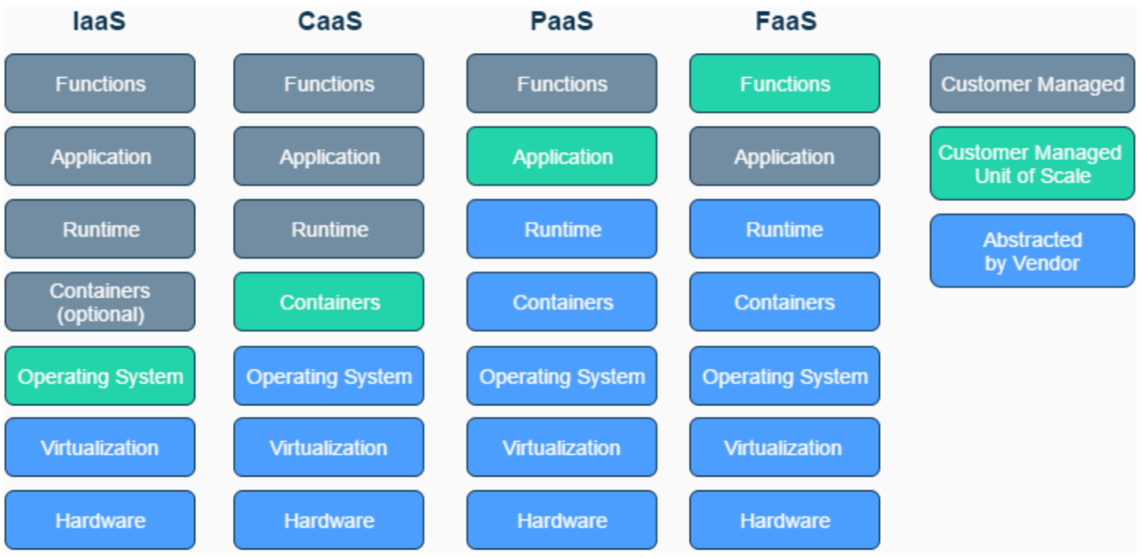


FIGURE 6. Differences in the level of abstraction for each type of services [10]

2.2 What is Serverless

Serverless is a broad term which can cause confusions for many individuals. At the heart of the term is a serverless architecture, which is a new methodology to architect a software system. It is a combination of BaaS (Backend-as-a-Service) and FaaS (Functions-as-a-Service). BaaS describes components in the system that are hosted in the infrastructure of third-party providers. The scalability and availability of those components are also guaranteed by them. Some examples are databases, messaging platforms, and user management. On the other hand, FaaS is a way to host the business logic of the system that will be triggered through some events in a fully-managed platform provided by a vendor. Also, it also delegates the tasks of deployment, maintenance, monitoring, provisioning and scaling to the vendor [2].

By applying the serverless architecture, organizations can build applications which are called “serverless applications” by utilizing “serverless services” provided by third parties. A service is defined to be serverless if it has the following five characteristics [2]:

Require no management of servers which host the service

There would probably be some servers or infrastructures needed to host the service, no matter whether companies are using a traditional architecture or a serverless architecture. Serverless here does not imply that there are no servers, it just means that they are hosted and maintained by someone else. AWS S3 (Amazon Simple Storage Service), which is a file storage service, is a case in point. Users communicate with the service through the APIs (Application Programming Interface) provided by the service to store their files without the need to know where and how those files are stored.

When it comes to monitoring the service, traditional metrics, such as the CPU usage are no longer needed since they would be controlled by vendors. Thus, measuring metrics that reflect how the service is used should now be the main focus of this architecture.

Scale seamlessly and automatically based on the load of the service

In the traditional server architecture, there are many challenging problems that need to be considered when maintaining the infrastructure. Firstly, engineers need to estimate resource types and numbers the application requires. To be more specific, they have to pick which machines, and which operating systems should be used and how strong the CPU is for the underlying servers. In addition, they also need to anticipate the expected load so that they can choose the right number of machines to be used. Secondly, they need to obtain the machines and prepare them so that those machines are ready to be put into production. Finally, they have to observe how the machines behave based on the load of the service so that they can scale the number of machines up and down in a timely manner. As a result, it would cost organizations a significant amount of time and human resources just to ensure that the service is running correctly. Usually, the safest way is to over-provision the resources which ensure that the service is always performed expectedly. Nevertheless, the solution would lead to the waste of resources as well as the increase in resource costs.

A serverless service has the ability to scale based on the load it receives. In other words, engineers no longer need to fulfill all the items in the list above to deploy the service in production.

AWS Lambda is a case in point. It is a service provided by AWS to run the business logic of the system which is represented as “functions” when external events happen. For example, if there is an HTTP request to our system, the Lambda service will automatically allocate a host which in turns spins up a container which will be used as an environment to execute the needed business logic. In addition, if there is another request coming in at the same time when the old request is still executing, the Lambda service will just allocate another host and repeat the process to serve the request. As a result, the service scales effortlessly no matter how many requests there are. After finishing processing those requests, the service will tear down those two allocated hosts which ensure there are no idle resources, thus reducing the cost.

Pay per request

The serverless service utilizes a bill-per-request model, meaning that users only pay for what they use. Thus, this model can reduce or increase the cost depending on the load of the system. For example, AWS Lambda only charges users for the execution time of the business logic per 100ms. This means that if the logic takes 4 seconds to complete, users pay for only that 4 seconds. Contradictory, traditional services, such as EC2, will charge users per hour, even only 10 seconds are used during that hour. Another example is AWS S3, which only counts the amount of data in GB stored in the service.

With this model, organizations can set up similar services to act as a back-up plan in other regions in case a disaster or an outage happens in one region without paying for any usages, thus increasing the availability of the system while keeping the cost to be the same.

However, this model does not always guarantee to bring benefits to all business use cases. Large systems with the constant or predictable load during the day should probably use the traditional architecture instead of the serverless architecture since it would be more economical in terms of infrastructure cost.

Define performance capabilities based on other characteristics

Generally, performance capabilities of a service are specified based on several servers the service is running on and how powerful those servers are. However, it is not the case for the serverless service. Serverless services use other attributes to determine how robust the service is. For instance, the AWS Lambda service allows users to specify how much RAM is needed for a function. By adjusting RAM, the CPU power is going to scale proportionally to that. Another example is AWS DynamoDB, which allows users to choose the provisional throughput in order to scale the underlying infrastructure based on that.

It is argued that serverless services are inferior to traditional services for this trait since users cannot specify the exact capability the service should have. However, it would bring justice to serverless services by emphasizing that today it is still the early days of serverless era. Thus, fine-grained configurations for those services are supposed to occur in the future.

Ensure high availability implicitly

A service is considered to have a high availability when it could operate normally or in the degraded state, even when one of its instances is malfunctioned. In the traditional architecture, a high availability is obtained by eliminating a single point of failure, meaning that each component in the system should have at least two instances running at the same time (e.g. multiple database instances or multiple web servers).

Needless to say, when it comes to serverless service, users no longer control the underlying infrastructure of a service, which leads to an assumption that the vendor who provides the service must ensure the high availability characteristic. For instance, if a serverless database is used then users can safely assume that a database cluster exists and is managed on their behalf. Similarly, for the serverless storage service, users' data must always be available even when the underlying nodes holding the data fail unexpectedly. Thus, it is the job of vendors to guarantee implicitly that serverless services will always be available.

2.3 Serverless Pros & Cons

There are no free lunches. Everything has two sides and serverless technologies are no exception. The most crucial factor when considering any technologies is that it has to fit business use cases. By analyzing the advantages and disadvantages of the serverless architecture, organizations can evaluate the viability of the serverless solution.

2.3.1 Advantages

Economical

Since the serverless architecture is built around serverless services, the billing model of those services will be the key factor in determining the cost of the system. Serverless services are billed per requests, meaning that there are no charges for idle capacity [1]. As a result, companies can save expenses by not paying for the idle time of the services while ensuring that they are always available when needed and the cost of the system can be seen as the reflection of the traffic going into the system.

Apart from service costs, operational costs are also reduced since service providers are in charge of managing the underlying infrastructure of the services.

Faster deployment

The serverless architecture encourages the microservices approach, meaning that the system is broken into smaller independent deployable services which are faster to deploy, thus improving the time to market as well as the ability to respond to any changes [11].

Scalability

Serverless services, which are the building blocks of the serverless architecture, are automatically scaled based on the actual use. AWS Lambda is a case in point. Each Lambda function contains some business logic which can be triggered if an event happens in the system. One example is that when a client (e.g. a website) makes an API call to an endpoint (e.g. API Gateway), it is considered to be an event and it can trigger the Lambda function which executes the logic. Assuming that there are five clients making API calls to the same endpoint at the same time, there will be five events which trigger five Lambda functions, all run at the same time. AWS S3 is another case in point. No matter whether a user needs to store 5GB or 5TB in the service, it can handle the requirement automatically by adding more disk spaces under the hood.

Besides, serverless services also ensure the availability, which is the ability to respond at least something to the request, and the fault tolerance by spanning the service to multi-region to avoid disasters in one region causing the entire service to be offline. There are no requirements in configuration or management to obtain those two features [11].

Operations overhead reduction

As said above, the operation responsibilities are now in the hand of service providers. To be more specific, organizations no longer need to provide, update, or monitor the servers. All problems related to hardware and server software are handled by the vendors. Besides, maintaining the tools, processes or on-call rotations to support the uptime of servers is unnecessary in this type of architecture [11]. As a result, companies can distribute more resources to the crucial parts of the business.

Easy transition

The conversion from the traditional architecture to the serverless architecture is straightforward if the codebase is already well-structured by following some design patterns, such as SOA (Services Oriented Architecture) or Clean Architecture. In that case, isolating the business logic of the application is all that is required for the transformation.

2.3.2 Disadvantages

Vendor lock-in

The serverless architecture is always made using serverless services, which are provided by cloud vendors. Undoubtedly, users can choose to adopt different services from different cloud vendors. However, the decision to go with different cloud vendors usually comes with the cost of complex configurations to integrate services from those vendors together. By using a set of services from the same provider, the smooth combination between services to build the complete architecture would be ensured.

Nevertheless, using a single vendor for the whole architecture would lead to the problem which is called “vendor lock-in”. Changing the vendor later, whether because of technical or business challenges, would be impossible without refactoring a large part of the codebase. That is why organizations should make careful considerations between diving into any cloud providers.

Uncontrolled environment

Since the responsibilities to manage the underlying infrastructure now belong to cloud vendors, users no longer have a transparent view about the environment. As a result, unexpected problems happening in the system are inevitable and the system should be designed to prepare for that. Power outages, disasters, and security breaches are some cases in point.

Unpredictable cost

The pay-per-execution model of serverless services is a great model targeted at systems that are not online all the time. Users do not have to pay for idle time, which turns out to be economical in the long run. However, due to the nature of the model, the cost can be varied since a load of a system is changing through the lifecycle of the product. At the beginning, there are only few users meaning that the cost for running the system will be minimal. As time goes by, the product becomes more popular, there will be more and more users using the product which leads to an enormous increase in the system cost. Moreover, during peak hours, there will also be a spike in the number of users, which makes the cost more and more unpredictable.

Local testing

A local test is the act of testing the behavior of the system in the local development environment. The system must work correctly in the local development before being deployed to the staging or production environment There are different types of testing, namely unit tests, integration tests, and e2e tests. Unit tests mean testing each component in isolation. Integration tests mean testing the interaction between components and e2e tests mean testing the full application flow as a user. This disadvantage is mostly applied to integration tests. With the traditional architecture, testing is trivial since every piece of software and dependencies needed to run on the production environment can be installed and used in the local environment. For instance, both environments can use the same MySQL database and web server with the difference only in configurations.

However, when it comes to the serverless architecture, things are not as straightforward as it seems anymore. In this type of architecture, serverless services are the core components which are provided by the cloud vendors. Since those services are closed-source, it is challenging to obtain the local installations to use in the development. Hence, integration tests between components in the system take a lot of effort to set up.

3 CLOUD PROVIDERS ANALYSIS

Since this thesis is about the serverless architecture, the analysis is carried out based on the serverless perspective, not the whole platform of each provider.

When considering which providers to rely on, companies should not only look at the serverless computing service to execute the core business logic code but also the whole serverless portfolio of the provider since serverless architecture is built around serverless services [11]. This is a crucial decision right from the beginning due to one disadvantage of serverless architecture which is vendor lock-in. Choosing the wrong one then changes to the right one later is going to cost a lot of resources.

A serverless platform consists of many services which can be combined to build a serverless application. A computing service, storage service, monitoring service, and message queue service are some cases in point. Each of the services should have the ability to scale automatically depending on the load of the application. The platform must meet the need of different types of customers, ranging from small startups to large enterprises. To fulfill that requirement, the features shown in the Figure 7 should be offered [11].

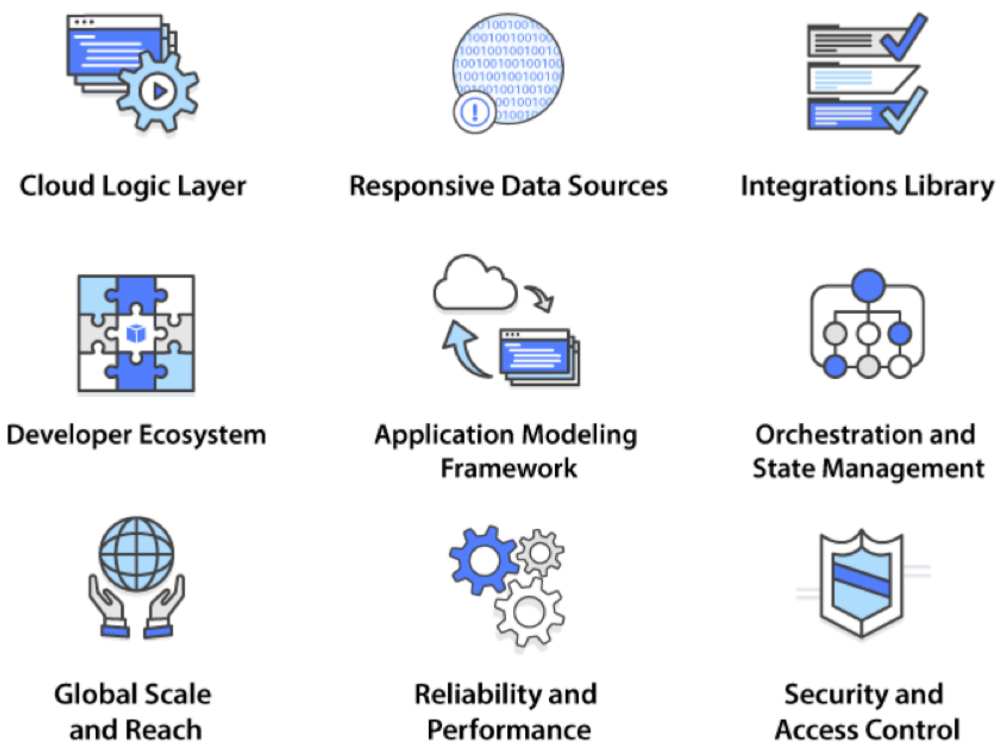

Cloud Logic Layer
Responsive Data Sources
Integrations Library
Developer Ecosystem
Application Modeling Framework
Orchestration and State Management
Global Scale and Reach
Reliability and Performance
Security and Access Control

FIGURE 7. Capabilities of a serverless platform [11]

As can be seen in the Figure 7, firstly, the minimum capability is the cloud logic layer which is in charge of running core business logic on demand. In addition, the ability to integrate that layer into various first-party or third-party services which act as event sources is critical as well. For instance, users might want to execute some business logic when an image is uploaded to the system. The provider should take responsibilities of that integration complexity and require users to perform as few configurations as possible. Secondly, easy-to-use integration libraries should also be a part of the platform [11]. Using the library can provide developers a trivial way to interact with the services, which makes it easy to adopt the new serverless model. Besides, having a stable developer ecosystem can support users during their day-to-day development [11]. Reusable solutions to some problems which are contributed by the community can improve the productivity of them.

Thirdly, the application modeling framework also needs to be supported to express the infrastructure as a code. By utilizing the framework, the deployment task now becomes declarative since users only need to provide which components exist in the application and cloud providers will manage the deployment. Besides, the orchestration and state management framework is also critical since it helps users to coordinate many short-lived functions in the cloud logic layer to become a long-running workflow that is suitable for various use cases [11].

Fourthly, in order to support a wide range of customers, including large enterprises which are multinational, the platform must offer a global scale meaning that there should be data centers located around the world. More than that, the reliability and performance of serverless services are also a key factor when considering cloud providers since the whole serverless system depends on them to operate properly [11].

Finally, the platform must have a built-in security and access control, such as virtual private networks, role-based and access-based permissions. The security of the system can be depicted as bread and butter of the organization thus cloud providers must strengthen it to complete their portfolio [11].

3.1 Serverless platform of cloud providers

There are many providers who joined the cloud game, but in this thesis, only three most popular ones are considered: Amazon Web Services, Microsoft Azure and Google Cloud Platform.

3.1.1 Amazon Web Services

AWS, which stands for Amazon Web Services, is a cloud service which is provided by Amazon. It offers different types of services which can be combined to build a software system. Some example services are computing services (e.g. Amazon EC2, AWS Lambda), storage services (e.g. Amazon S3), database services (e.g. Amazon RDS), and messaging services (e.g. Amazon SQS, Amazon SNS) [12]. Other core services are described in the Figure 8.

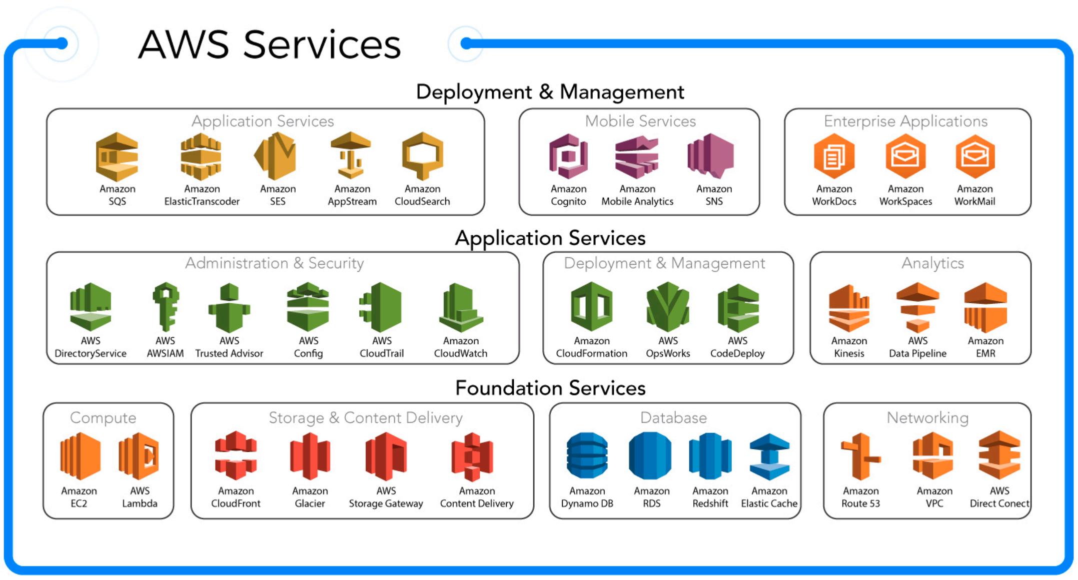


FIGURE 8. Services offered by AWS [16]

The existence of AWS Lambda marks the beginning of the serverless era for AWS. It was introduced in 2014 and has attracted countless organizations to start transforming their system to the serverless architecture since then. Netflix's adoption of serverless using AWS Lambda is a popular case in point [13].

3.1.2 Microsoft Azure

Microsoft Azure, which was introduced in February 2010, is the answer of Microsoft for the cloud war. They offer many cloud services which can be leveraged to build a software system. Some example services are computing services (e.g. Azure Functions, Azure Virtual Machines), database services (e.g. Azure CosmosDB, Azure Database for MySQL), storage services (e.g. Azure Blob storage), messaging services (e.g. Azure Queue Storage, Event Grid) [14]. Other core services are described in the Figure 9.

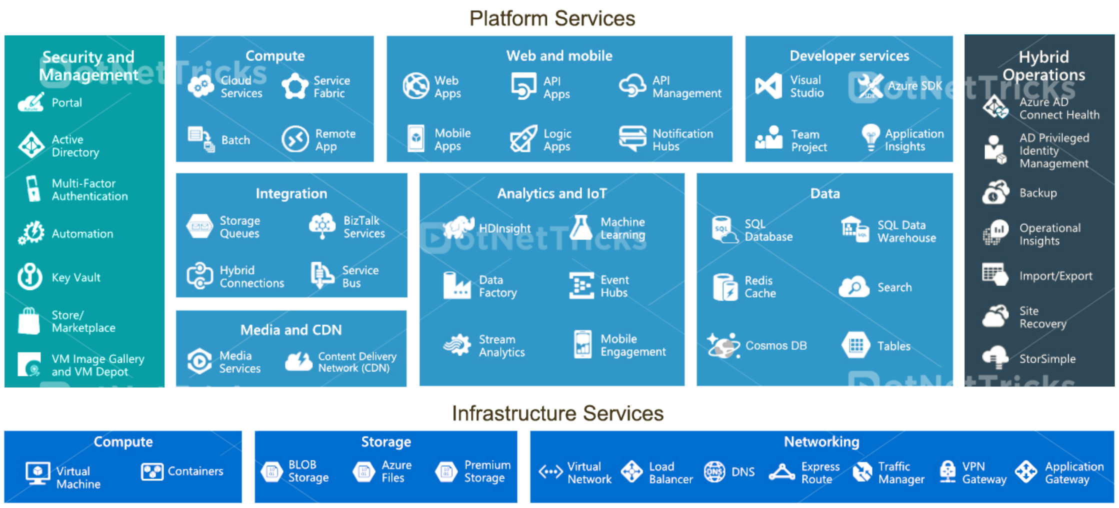


FIGURE 9. Services offered by Azure [17]

Just like AWS, the existence of Azure Functions in 2016 marks the beginning of the serverless era for Microsoft.

3.1.3 Google Cloud Platform

GCP, which stands for Google Cloud Platform, was introduced in April 2008. They offer a wide range of services just like Microsoft Azure and AWS. Some example services are computing services (e.g. Google Cloud Functions, Google Compute Engine), database services (e.g. Google Cloud Datastore, Google Cloud SQL), storage services (e.g. Google Cloud Storage), messaging services (e.g.Google Cloud Pub/Sub) [15]. Other core services are described in the Figure 10.

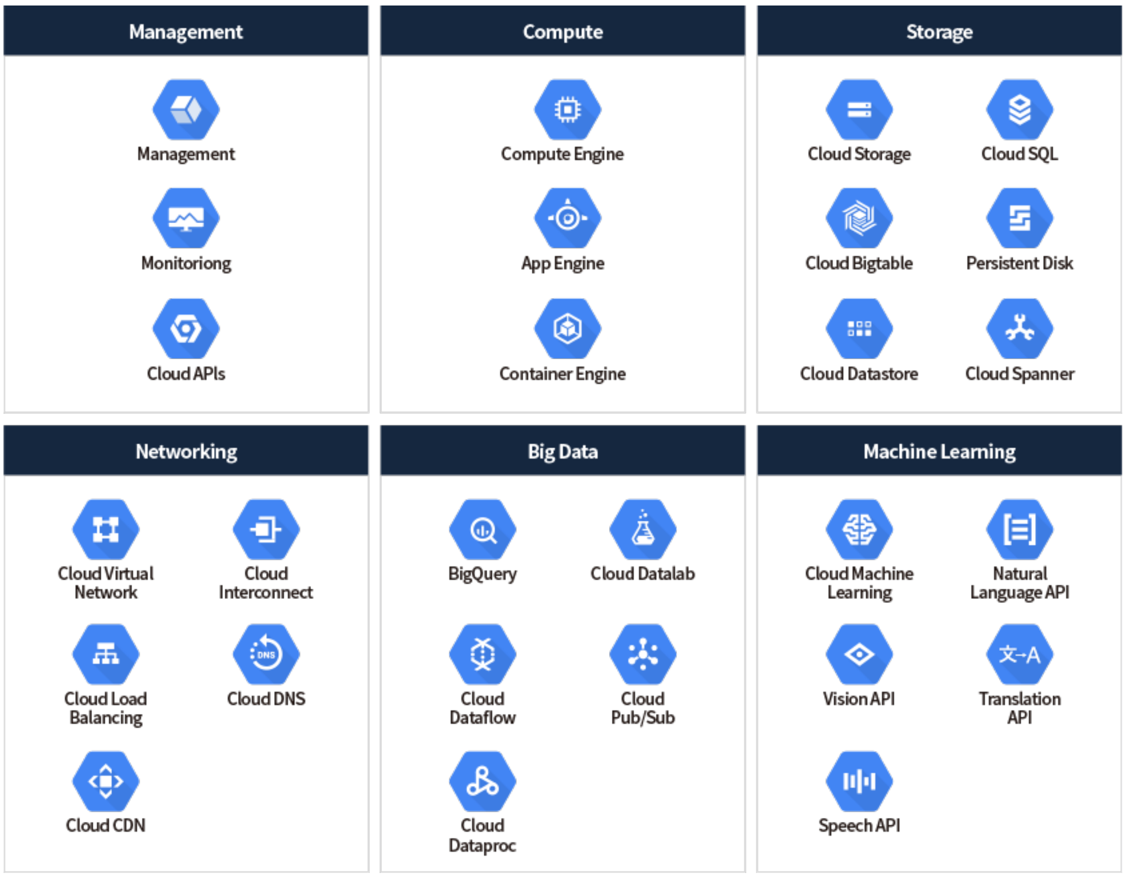


FIGURE 10. Services offered by GCP [18]

Just like AWS and Microsoft Azure, the existence of Google Cloud Functions in 2017 marks the beginning of the serverless era for Google.

3.2 Comparison of serverless offers

There are many factors that need to be analyzed when choosing to utilize the serverless platform of a cloud provider [13]. No cloud provider is dominant at every factor which leads to the fact that the decision depends on the nature of the system and the organization.

Pricing Models

When it comes to a cloud logic layer, which is the main service in the serverless platform, all cloud providers follow the pay-per-use model, but the cost is varied per second.

AWS Lambda offers a free-tier plan which includes 1 million requests and 400000GB-seconds of computing time per month. After the free-tier, the computing time is charged $0.00001667/GB-s. Allocated memory and CPU are billed together.

Azure offers the same free-tier plan as AWS, but after the free-tier, the billing is $0.000016/GB-s. Azure also differs from AWS to the extent that they charge consumed memory instead of allocated one.

GCF offers 2 million requests per month and the same computing time which is similar to Azure and AWS for their free-tier plan. After that, the price is $0.000004 per request including network traffic, which is considered to be higher than others regarding the length of time a function runs versus the number of requests. They also bill memory and CPU separately.

To sum up, in this factor, AWS Lambda offers more reasonable pricing than others.

Supported programming languages

AWS supports a wide range of languages for writing their cloud logic layer, which is AWS Lambda. The supported languages are JavaScript, Python, Golang, Java, C#, Visual Basic and F#.

Azure Functions supports JavaScript, C#, F#, Python, PHP, Bash, Batch and PowerShell.

Google Cloud Functions only supports JavaScript for now.

As can be seen clearly, AWS and Azure are more flexible since they allow a list of different languages.

Trigger types

Trigger types are different kinds of events that can be used to invoke the function.

AWS Lambda has a wide range of triggers. Some examples are HTTP triggers (e.g. when an API request reaches in an API Gateway), a file-based trigger (e.g.

when a file is uploaded to S3), database triggers (e.g. when a data is inserted into DynamoDB).

Azure Functions has the same trigger types with AWS Lambda.

Google Cloud Functions offers fewer types, such as HTTP triggers and messaging triggers (e.g. when a message is sent to a topic in Cloud Pub/Sub)

In conclusion, AWS Lambda and Azure Functions offer many more trigger types, allowing different kinds of combinations for the event-driven architecture.

Maximum execution time and concurrency

Execution time means how long a function can run before it is automatically timed out. Concurrency denotes the ability to run several functions in parallel.

AWS allows 1,000 concurrent functions at any given point of time. The maximum execution time for a function is 15 minutes.

Azure offers unlimited concurrent functions per application. The maximum execution time for a function is 5 minutes by default, but it can be upgraded to 10 minutes.

GCP defines the limit per trigger types. To be more specific, the HTTP trigger type supports unlimited concurrent functions. For other types, the limit is 1,000 functions at any given point of time. The maximum execution time is 1 minute by default and 9 minutes after being upgraded.

To sum up, if concurrency is important, then GCP and Azure would be best choices. However, if long execution time is necessary, then AWS would be a safe option.

Deployment methods

With the introduction of the Serverless framework, all three cloud providers are the same in this category. Deployment is straightforward and trivial with just one CLI command.

Monitoring

A monitoring service is crucial in the serverless architecture because of the server abstraction. Developers cannot control the underlying infrastructure which is why the monitoring service is everything they can rely on to keep track of the health of the system.

All providers seem equal in this category since they all have their monitoring service. AWS has Amazon CloudWatch, Microsoft Azure has Microsoft Monitor and GCP has Stackdriver.

3.3 AWS in depth

There are many services provided by AWS that can be combined to build a serverless application, namely:

- Compute: AWS Lambda
- APIs: Amazon API Gateway
- Storage: Amazon Simple Storage Service (Amazon S3)
- Databases: Amazon DynamoDB
- Messaging: Amazon Simple Notification Service (Amazon SNS) and Amazon Simple Queue Service (Amazon SQS)
- Deployment: Amazon CloudFormation
- Monitoring: Amazon CloudWatch

3.3.1 AWS Lambda

Among those services, AWS Lambda is the most crucial one since it is the core of a serverless application. By achieving an in-depth understanding of Lambda, the serverless architecture will become more accessible to developers.

AWS Lambda is a FaaS service. It is the cloud logic layer of a serverless application. It provides computing functionality under the form of functions. Those functions can be triggered by various events which happen on AWS or third-party services. The event sources for those events are usually AWS services (e.g. Amazon S3 and Amazon API Gateway). Functions will execute in parallel if there are many concurrent events [19]. In addition, it follows the pay-per-request model as discussed in the previous section, meaning that there are no additional charges if there are no events.

As shown in the Figure 11, each Lambda function contains the application logic as code, the configuration for the function and at least one event source which emits events that Lambda will respond to. Amazon S3 as an event source is a case in point. Lambda can be configured so that whenever a file is uploaded to S3, a function will be run the information about the file. The logic for that function can be to compress the file, for example.

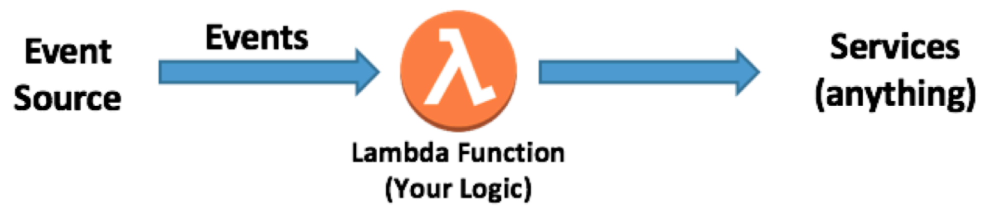


FIGURE 11. The simplified architecture of a running Lambda function [19]

What happens under the hood is that whenever an event is emitted from the event source which is attached to a Lambda function, that Lambda function will be initialized. The initialization process includes spinning up the execution environment for the function, which is usually a container inside a virtual machine. After the container is up, software dependencies for the programming language that are used by the function will be installed. Then, the code for application logic, which is usually uploaded on Amazon S3 by developers, is downloaded. Finally, the code will be executed with the parameters depending on the type of the event.

One important consideration when writing code on AWS Lambda is that the logic should not make any assumptions about the state of the function [19]. The container running the first function might be different from the one which runs the second function, which results in a completely different state between two functions.

Event sources for a Lambda function can follow either a push model or a pull model. In the push model, a Lambda function will be executed whenever an event happens in the event source. On the other hand, in the pull model, Lambda will poll the event source periodically and combine several new events into one function invocation. The Table 1 below shows different types of event sources and their trigger condition:

TABLE 1. Examples of event sources [19]

Event Source	Invocation Model	Example Trigger Condition
Amazon S3	Push	Whenever an object is created or removed in S3, a Lambda function can be triggered to run with the information about that file.
Amazon API Gateway	Push	Whenever an API request comes in, a Lambda function can be triggered to run with the information about that request and it is expected to return a response to the client.
Amazon SNS	Push	Whenever there is a message that is published to an SNS topic, a Lambda function can be triggered to run

		with the content of the message.
Amazon SQS	Pull	A Lambda function can be configured to poll the SQS queue periodically to check for new messages and perform some logic with the content of those messages.
Amazon DynamoDB	Pull	A Lambda function can be configured to poll the DynamoDB stream to check for any updates (e.g. new rows inserted, old rows deleted) since the last batch and perform some logic with the content of those rows.
Amazon CloudWatch Event	Push	Whenever there is a change in the state of a resource, a Lambda function can be triggered to run with the information about the change.

3.3.2 Amazon API Gateway

The Amazon API Gateway is a serverless service managed by AWS providing the ability to create REST and WebSocket APIs for a system. The service acts as the entry point to the system and is heavily used in almost every serverless application today as an event source together with AWS Lambda. The following terminologies are crucial when working with the API Gateway [20]:

1. Resource: Each resource is an URL endpoint with its path. For instance, api.foo.com/bar is a resource.
2. Method: A method consists of a resource path and an HTTP verb. For example, GET /bar is a method.
3. Method Request: A method request consists of the method itself together with URL query string parameters and HTTP request headers.
4. Integration Request: Defines the backend target to be used with the method. Lambda integration is a case in point where requests are forward to a Lambda function. Request mappings to transform the request body to the appropriate parameters for the backend target are also performed at this point.
5. Integration Response: Defines the response mapping between the backend target and the API Gateway. To be more specific, the API Gateway can transform the response from the backend target to be the understandable one for the client.
6. Method Response: A method response consists of response types, their headers, and their content types.
7. Model: A model defines the shape of the request body. A model can be used to perform validation against the body. The model is written in the JSON schema format.
8. Stage: A stage is used to separate different deployment environments so that they can exist in parallel. The development stage and production stage are cases in point.

3.3.3 Amazon S3

Amazon S3 is a serverless storage service provided by AWS. Users can put an unlimited number of files in S3 for a persistent storage. However, one file is limited to the maximum size of 5TB. AWS ensures that Amazon S3 meets the durability of 99.999999999% and the availability of 99.99%.

One example use case of Amazon S3 is to store AWS Lambda sources code of a serverless application. In addition, S3 can also be configured as an event source for AWS Lambda so that whenever there is a change, such as a file is uploaded, a Lambda function is triggered to respond to that event.

3.3.4 Amazon DynamoDB

Amazon DynamoDB is a serverless NoSQL database provided by AWS. It is well known by its highly scalable feature which can handle millions of requests per second. Since it is a serverless database, users can increase its capacity and throughputs by modifying RCU (Read Capacity Unit) and WCU (Write Capacity Unit).

The unit model of DynamoDB are tables with a partition key and an optional range key. A partition key is used to divide the data into different partitions. Each partition will hold a portion of the total data. A range key is used to sort the data inside a partition. By combining the partition key and the range key, powerful queries can be executed to extract needed data from the database. Another option is to get all the data in the database regardless of the keys. However, it is not a common use case since getting all the data is an expensive operation regarding the performance and the cost. With that being said, understanding the query patterns of the data before setting up the database is essential.

3.3.5 Amazon SNS

Amazon SNS is a serverless pub/sub messaging service provided by AWS. It allows publishers to send messages to subscribers. In this service, publishers do not know about subscribers and they do not interact with each other directly but through a thing called “topics”. Publishers can send messages to a topic from which the messages will be delivered to the subscribers of the topic. The Figure 12 provides a visual overview on how the service works.

This service encourages asynchronism, which is the ability of a component to communicate with other components in the system asynchronously, without waiting for the response.

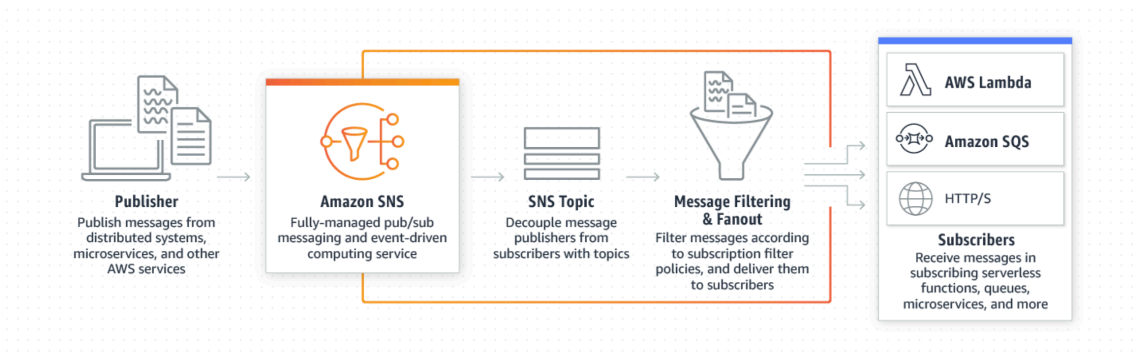

Publisher
Publish messages from distributed systems, microservices, and other AWS services
Amazon SNS
Fully-managed pub/sub messaging and event-driven computing service
SNS Topic
Decouple message publishers from subscribers with topics
Message Filtering & Fanout
Filter messages according to subscription filter policies, and deliver them to subscribers
AWS Lambda
Amazon SQS
HTTP/S
Subscribers
Receive messages in subscribing serverless functions, queues, microservices, and more

FIGURE 12. How AWS SNS works [21]

3.3.6 Amazon SQS

Amazon SQS is a serverless messaging queuing service provided by AWS. Messages are sent by senders to the message queue from where receivers continuously poll to get the messages. After processing a message, the receiver needs to delete it from the queue so that it will not be processed twice.

This service is somehow similar to Amazon SNS when observing from the outside since they are both a messaging service, but they are much different in their nature. Amazon SNS is a push model in which messages are pushed to the subscribers and there can be various subscribers processing the same message. On the other hand, Amazon SQS is a poll model where the receivers need to poll the queue to get the messages and process them. A message cannot be received by multiple receivers at the same time. The message should be deleted after the receiver finishes processing it.

3.3.7 Amazon CloudWatch

Amazon CloudWatch is a serverless monitoring service provided by AWS. By utilizing CloudWatch, developers can obtain statistics about the system health in near real-time. Logs, metrics, and events can be collected from the system and are sent to Amazon CloudWatch. For example, when integrating with AWS Lambda as an event source, logs emitted from Lambda functions can be sent to CloudWatch for analyzing purpose.

In addition, it also provides CloudWatch Alarms which can be triggered when the predefined condition is violated. For instance, whenever a Lambda function throws an error, an alarm can be triggered to send a message with information about the function to a topic in Amazon SNS, which can be attached to another Lambda function to perform some logic based on the error.

3.3.8 Amazon CloudFormation

Amazon CloudFormation is a service provided by AWS that can help engineers to describe the components in their system in a declarative way. CloudFormation encourages companies to adopt the Infrastructure-As-A-Code model. In this type of model, components in the system, such as databases and computing resources, are described in a template as code. The template can be committed to a version control platform so that any engineer in the team can explore and contribute to the infrastructure without the need for consulting the engineer who maintains the infrastructure. The template can then be deployed to AWS in which it becomes a CloudFormation stack. One AWS account can have many stacks and each stack represents a set of system components. To update a stack, a changeset needs to be created by modifying the template and uploading it to AWS again. A stack can be deleted manually through an AWS console or programmatically through AWS SDK which will also delete all the components associated with the stack.

4 A SIMPLE SERVERLESS APPLICATION

In the practical part, a serverless application will be implemented to demonstrate the ability to create and develop serverless applications on AWS. The motivation for the application comes from the daily life need of the author. The author needs to track the expiration date of meat boxes he bought. Instead of using pen and paper, he would like to leverage his Google Home device to perform this task. To be more specific, Google Home can answer two different kinds of requests from the author: “Hey Google, meat XX (will expire/expiration date) on 10th of December)” and “Hey Google, what is the nearest meat expiration date?”. The high-level architecture will be presented first, which will be followed by the brief introduction of technologies used in the application and the detailed user flow.

4.1 High-level architecture

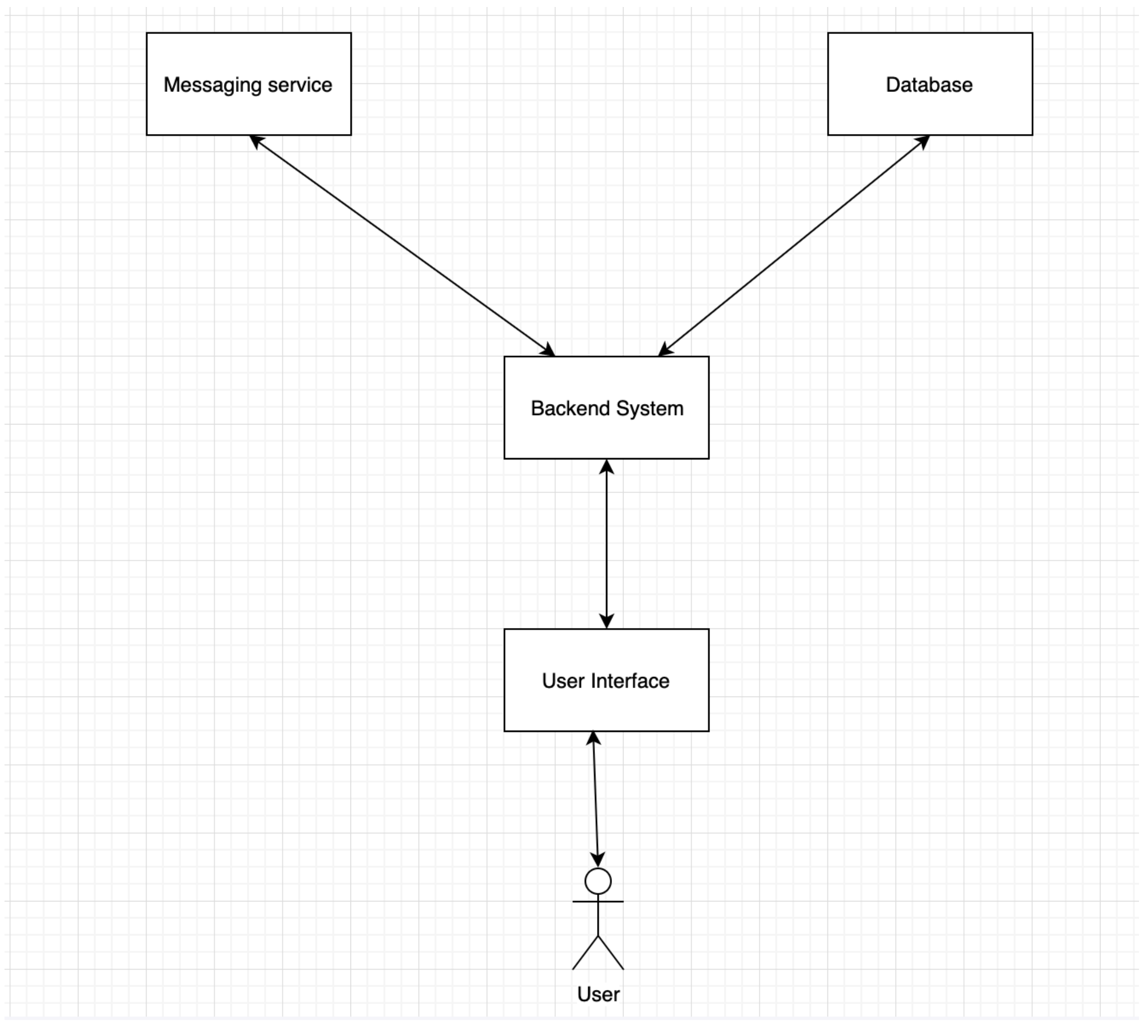


FIGURE 13. High-level architecture

As shown in the Figure 13, the high-level architecture of the application is generic enough to be adapted to applications following traditional architectures as well. Main components in the architecture are User Interface, Backend System, Messaging Service, and Database. Initially, a user will interact with the user interface, which is in charge of communicating with the backend system to save or fetch data from the database. In addition, the backend system also connects with the messaging service to provide asynchronous communication. In this demo application, the user interface is a Google Home Mini device, the backend system is the AWS API Gateway + AWS Lambda, the messaging service is Amazon SNS and the database is DynamoDB. The application is written in the Go programming language and is deployed on AWS using the Serverless framework.

4.2 Technologies introduction

Since the Amazon API Gateway, AWS Lambda, Amazon DynamoDB and Amazon SNS have been discussed in the previous section, only the Google Home Mini device, Serverless framework and Go programming language are introduced in this section.

4.2.1 Google Home Mini Device

Google Home Mini, which is depicted in the Figure 14, is a smart speaker developed by Google. It provides the ability for users to interact with devices using commands through Google Assistant. Google Assistant is powered by Machine Learning under the hood, which offers powerful capabilities to respond to different kinds of complex commands from users. Some example use cases include asking for today’s weather or turning on the TV and music. It can also be integrated into various electronic devices to perform home automation.

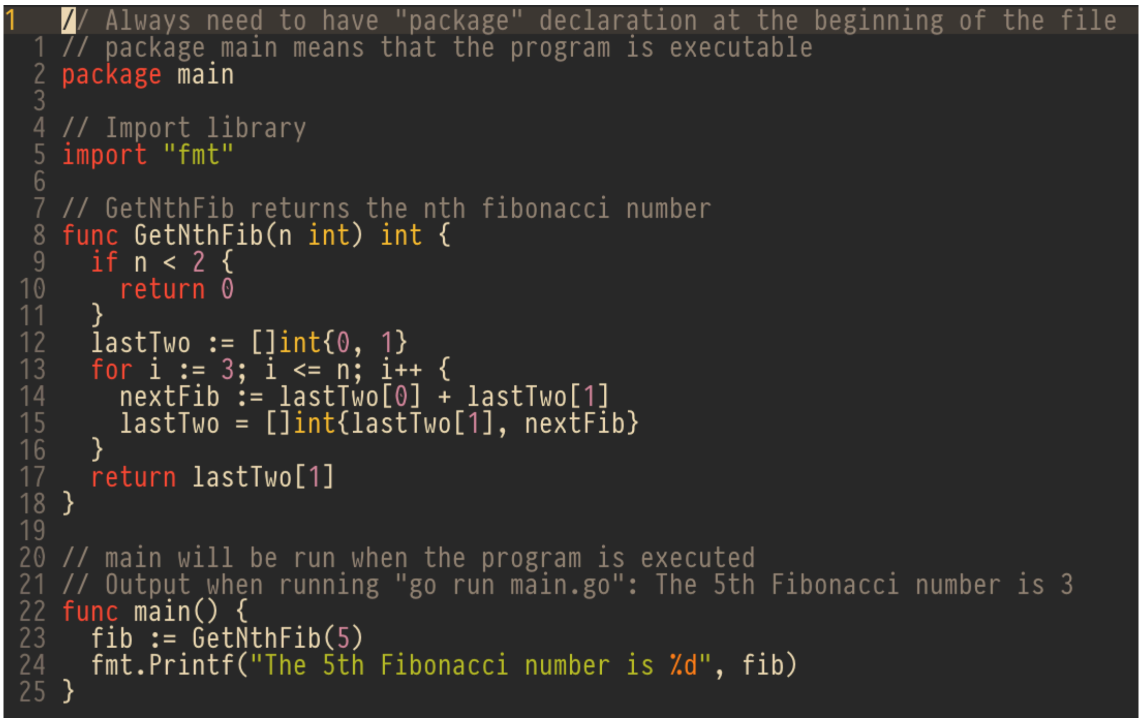
FIGURE 14. The Google Home Mini device [22]

4.2.2 Dialogflow

Dialogflow is a Google service that can be used to create custom functionalities for Google Assistant which is the underlying technology of Google Home devices. At the heart of the service is an agent, which represents an application. Each agent can have many intents which represent user requests, such as play a video or make an order. One intent can have many entities, which can be extracted to get information about a user request. Finally, each intent can be associated with a fulfillment which can be used to provide a custom response for a request. In this thesis, the agent is the meat expiration tracking service. The service will have two intents, which are the “ask meat expiration date” and the “save meat expiration date”. Inside those intents, there will be two entities, namely the “meat name” and the “expiration date”. The fulfillment will be a custom endpoint provided by the backend system to respond to the requests.

4.2.3 Go Programming Language

Go is a statically typed, compiled programming language developed at Google. The goal of Go is to have a static typing and run-time speed like C++, readability and simplicity like Python, and JavaScript with efficient multiprocessing. The most critical feature of Go is the ability to compile the code to a single executable binary without any dependency, which makes the deployment process effortlessly. The Figure 15 shows an example of a Go program.

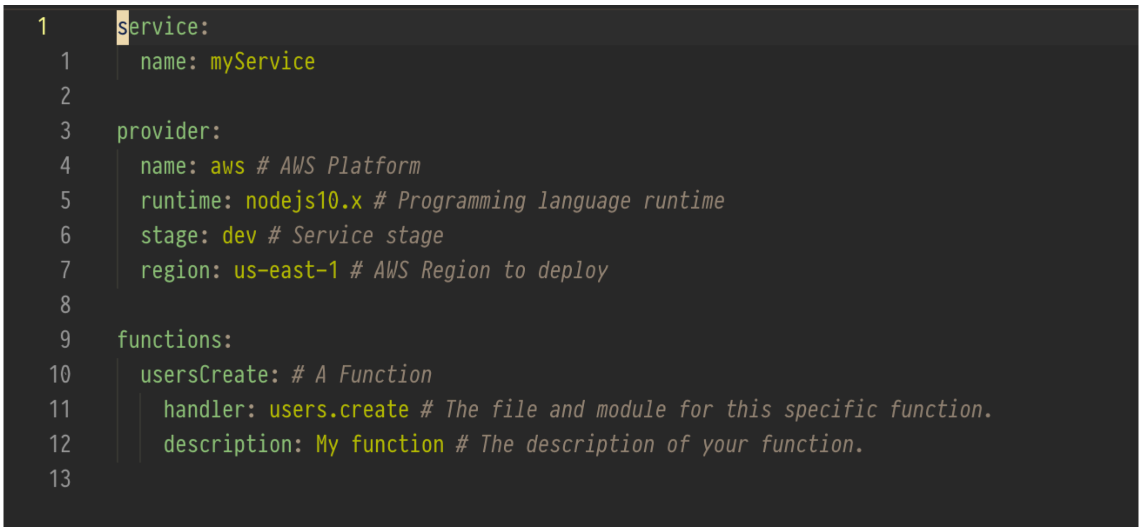


```go
// Always need to have "package" declaration at the beginning of the file
// package main means that the program is executable
package main

// Import library
import "fmt"

// GetNthFib returns the nth fibonacci number
func GetNthFib(n int) int {
  if n < 2 {
    return 0
  }
  lastTwo := []int{0, 1}
  for i := 3; i <= n; i++ {
    nextFib := lastTwo[0] + lastTwo[1]
    lastTwo = []int{lastTwo[1], nextFib}
  }
  return lastTwo[1]
}

// main will be run when the program is executed
// Output when running "go run main.go": The 5th Fibonacci number is 3
func main() {
  fib := GetNthFib(5)
  fmt.Printf("The 5th Fibonacci number is %d", fib)
}
```

FIGURE 15. An example of a Go program

4.2.4 Serverless Framework

The Serverless framework is an open-source deployment framework for serverless applications. Components and configurations of an application can be written declaratively in a file called “serverless.yml”, which is shown in the Figure 16. After having the “serverless.yml” file, the application can be deployed with just one CLI command which vastly improves the developer experience. Since the framework is provider-agnostic, every serverless cloud platform can be integrated and can take advantage of it.



```yaml
service:
  name: myService

provider:
  name: aws # AWS Platform
  runtime: nodejs10.x # Programming language runtime
  stage: dev # Service stage
  region: us-east-1 # AWS Region to deploy

functions:
  usersCreate: # A Function
    handler: users.create # The file and module for this specific function.
    description: My function # The description of your function.

```

FIGURE 16. An example serverless.yml file

The example above will create an AWS Lambda function which is written in NodeJS and deployed in the us-east-1 region.

4.3 Detailed user flow

Flow 1: Save meat expiration date

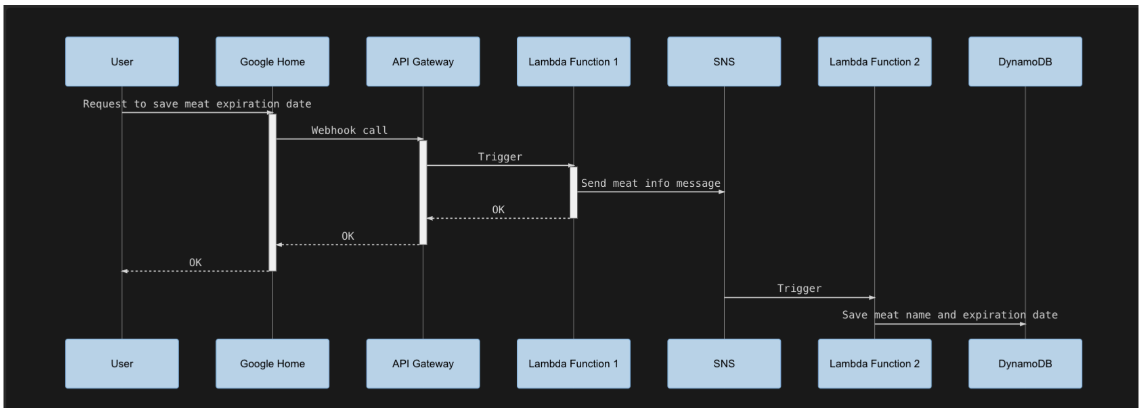


FIGURE 17. The user flow to save the meat expiration date

As demonstrated in the Figure 17, a user will make a voice command to a Google Home device, such as: “Hey Google, meat XX will expire on 10th of December”. After receiving the command, the device will extract the information such as a meat name and expiration date from the command and send a POST request with that information to the webhook endpoint URL provided by the API Gateway, for example, https://api.example.com/meat-request-handler. Then, the API Gateway will trigger a Lambda function and pass command information as arguments to the function. Next, the Lambda function will create a message containing the meat name and expiration date and act as a publisher to send it to an SNS topic. Lastly, a subscriber of that SNS topic, which is a Lambda function, will receive the message and attempt to save the meat name with its expiration date to DynamoDB. The reason for choosing asynchronous communication through Amazon SNS is that the data persistence process does not need to happen immediately. Also, by leveraging Amazon SNS, different type of business logic can be performed on the information about the meat name and its expiration date through attaching more Lambda functions later if needed.

Flow 2: Ask for nearest expiring meat

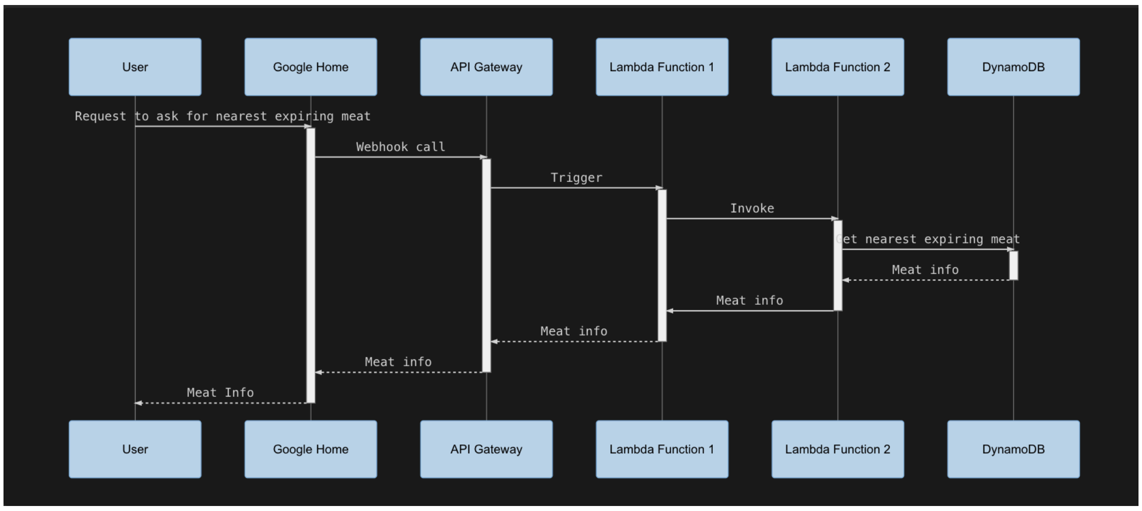


FIGURE 18. The user flow to ask for nearest expiring meat

As demonstrated in the Figure 18, a user will make a voice command to a Google Home device, such as: “Hey Google, when is my nearest meat expiration date?”. After receiving the command, the device will extract the information, such as the meat name and expiration date from the command and send a POST request with that information to the webhook endpoint URL provided by the API Gateway, for example, https://api.example.com/meat-request-handler. Then, the API Gateway will trigger a Lambda function and pass command information as arguments to the function. Next, the Lambda function will make the query to DynamoDB to figure out which meat will expire next. After getting the query result from DynamoDB, the function will respond to the API Gateway, which in turn responds to the Google Home device. In the end, the user will hear the response from the device which is: “The nearest expiring meat is pork on 12th of December 2019”.

The full source code is available at https://gitlab.com/ntuandung93/joelin.

5 CONCLUSION

Serverless architecture appears to be a solid choice for organizations. It can cover various use cases while keeping the advantages outweigh the disadvantages. Infrastructure costs can now be utilized for other areas which can boost the core product value. From the author’s perspective, the serverless architecture would probably have strong potentials in the future and can become a standard way of developing cloud-based systems.

However, adopting the serverless model would require an extensive refactor process. Companies need to consider and split the resources wisely to balance between core business values and internal restructurings. The recommendation from the author is to reconstruct the codebase to follow the service-oriented architecture first. Then, converting each service to the serverless architecture incrementally.

REFERENCES

1. Wagner, T. 2017. Optimizing Enterprise Economics with Serverless Architectures. Amazon Web Services
2. Roberts, M. 2017. Defining Serverless – Part 1. Date of retrieval 4.6.2019 https://blog.symphonia.io/defining-serverless-part-1-704d72bc8a32
3. Lahiri, M. 2017. The evolution from servers to functions. Date of retrieval 10.6.2019 https://read.acloud.guru/the-evolution-from-servers-to-functions-21833b576744
4. Reed, J. 2018. Physical Servers vs Virtual Machines: Key Differences and Similarities. Date of retrieval 13.6.2019 https://www.nakivo.com/blog/physical-servers-vs-virtual-machines-key-differences-similarities/
5. Fiber Optic Solutions. 2017. Server Rack Choice: How to Make the Right Decision? Date of retrieval 15.6.2019 http://www.fiber-optic-solutions.com/server-rack-choice-right-decision.html
6. Network Computing Solutions. NCS-Server-Virtualization. Date of retrieval 20.6.2019 http://www.ncs-grp.com/project/project-three/ncs-server-virtualization/
7. Lardinois, F. 2016. WTF is a container? Date of retrieval 22.6.2019 https://techcrunch.com/2016/10/16/wtf-is-a-container/
8. Rubens, P. 2017. What are containers and why do you need them? Date of retrieval 25.6.2019 https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.html
9. Watts, S. & Raza, M. 2019. SaaS vs PaaS vs IaaS: What's The Difference and How To Choose. Date of retrieval 28.6.2019 https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/
10. McKim, J. 2016. Abstracting the Back-end with FaaS. Date of retrieval 30.6.2019 https://serverless.zone/abstracting-the-back-end-with-faas-e5e80e837362

11. Bashir, F. 2018. What is Serverless Architecture? What are its Pros and Cons? Date of retrieval 2.7.2019 https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9
12. Sharma, H. 2019. What is AWS? – An Introduction to AWS. Date of retrieval 4.7.2019 https://www.edureka.co/blog/what-is-aws/
13. Lobastov, I. 2019. Comparing Serverless Architecture Providers: AWS, Azure, Google, IBM, and Other FaaS Vendors. Date of retrieval 5.7.2019 https://dzone.com/articles/comparing-serverless-architecture-providers-aws-az
14. Microsoft Azure. Azure Products. Date of retrieval 5.7.2019 https://azure.microsoft.com/en-in/services/
15. Wikipedia. Google Cloud Platform. Date of retrieval 6.7.2019 https://en.wikipedia.org/wiki/Google_Cloud_Platform
16. Guillermo 2018. Why to choose AWS Cloud for your Web Application? Date of retrieval 8.7.2019 https://www.clickittech.com/aws/why-aws-cloud/
17. Chauhan, S. 2017. What is Microsoft Azure? – An Introduction to Azure. Date of retrieval 11.7.2019 https://www.dotnettricks.com/learn/azure/getting-started-with-microsoft-azure-platform
18. Bespin Global. Google Cloud Platform. Date of retrieval 12.7.2019 https://en.bespinglobal.com/product-solutions/google-cloud-platform/
19. Baird, A. & Huang, G. & Munns, C. & Weinstein, O. 2017. Serverless Architecture with AWS Lambda. Amazon Web Services
20. QwikLabs. Introduction to Amazon API Gateway. Date of retrieval 13.7.2019 https://www.qwiklabs.com/focuses/269?parent=catalog
21. AWS. Amazon Simple Notification Service. Date of retrieval 15.7.2019 https://aws.amazon.com/sns/
22. Fingas, J. 2017. Google Home Mini is a basic $49 smart speaker. Date of retrieval 20.7.2019 https://www.engadget.com/2017/10/04/google-home-mini/