SERVERLESS COMPUTING

A Seminar Report

Submitted by

BONKURI BHAVANI PRASAD
(16BQ1A0523)

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

IN
COMPUTER SCIENCE & ENGINEERING

At

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY
Affiliated to JNTUK, Approved by AICTE, Accredited by NBA & NAAC
Nambur (v), Peda kakani(P), Guntur Dt.- 522508

July, 2020

CERTIFICATE

This is certify that the seminar titled “SERVERLESS COMPUTING” is a bonafide work carried out by Mr. Bonkuri Bhavani Prasad Regd.no 16BQ1A0523 in a partial fulfillment for the award of AMIETE Computer Science during the year 2020. It is certified that all the corrections/ suggestions indicated for internal assessment have been incorporated in the report deposited in the VVIT library. The seminar report had been approved as it satisfies the academic requirement in respect of seminar work presented for the award of B.Tech Bachelor Engineering degree.

Name of the Student: Bonkuri Bhavani Prasad

Registration No: 16BQ1A0523

Name and signature of external guide

Seminar I/C **HoD, CSE**

ABSTRACT

Serverless computing has emerged a new compelling paradigm for the deployment of applications and services. It represents an evolution of cloud computing models, abstractions, and platforms. And is a testament to the maturity and wide adoption of cloud techonologies. In this I will explain the architecture, methods to run application serverless, and applications of serverless, advantages and disadvantages of serverless computing.

With most people still trying to get their head around containerized microservice applications, Serverless Computing has yet to become mainstream. However, the never-ending quest for more speed in the application delivery lifecycle makes Serverless Computing more desirable. Before those advantages can be fully realized, though, additional tooling is needed to make it easier to manage, monitor and debug Serverless Applications.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The term serverless computing is a misnomer as the technology uses servers, only they're maintained by the provider. Serverless computing is considered a new generation of Platform as a Service, as in this case, the cloud service provider takes care of receiving client requests and responding to them, monitoring operations, scheduling tasks, and planning capacity. Thus, developers have no need to think about server and infrastructure issues and can entirely concentrate on writing software code.

Serverless computing is also sometimes called **Function as a Service** or event-based programming, as it uses functions as the deployment unit. The event-driven approach means that no resources are used when no functions are executed or an application doesn't run. It also means that developers don't have to pay for idle time. In addition, a serverless architecture ensures auto-scaling, allowing applications to provide users with services in spite of increasing workload.

Thanks to all these features, serverless computing is considered a cost-saving, resource-limited, and fault-tolerant approach to software development. This novel approach is already provided by such industry giants as Amazon, Google, IBM, and Microsoft.

Serverless computing is a cloud computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of machine resources. Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity. It can be a form of utility computing.

Serverless computing can simplify the process of deploying code into production. Scaling, capacity planning and maintenance operations may be hidden from the developer or operator. Serverless code can be used in conjunction with code deployed in traditional styles, such as microservices. Alternatively, applications can be written to be purely serverless and use no provisioned servers at all.

This should not be confused with computing or networking models that do not require an actual server to function, such as peer-to-peer (P2P).

Serverless is the native architecture of the cloud that enables you to shift more of your operational responsibilities to AWS, increasing your agility and innovation. Serverless allows you to build and run applications and services without thinking about servers. It eliminates infrastructure management tasks such as server or cluster provisioning, patching, operating system maintenance, and capacity provisioning. You can build them for nearly any type of application or backend service, and everything required to run and scale your application with high availability is handled for you.

Before we get into what FaaS is, it's important to understand another term — **serverless computing**. Serverless computing is a cloud computing model which aims to abstract server management and low-level infrastructure decisions away from developers. In this model, allocation of resources is managed by the cloud provider instead of the application architect, which can bring some serious benefits. In other words, serverless aims to do exactly what it sounds like — allow applications to be developed without concerns for implementing, tweaking, or scaling a server (at least, to the perspective of a user).

Functions as a Service

FaaS is a relatively new concept that was first made available in 2014 by hook.io and is now implemented in services such as AWS Lambda, Google Cloud Functions, IBM OpenWhisk and Microsoft Azure Functions. It provides a means to achieve the serverless dream allowing developers to execute code in response to events without building out or maintaining a complex infrastructure.

FaaS Advantages

1. Fewer developer logistics — server infrastructure management is handled by someone else.

2. More time focused on writing code / app specific logic — higher developer velocity.

3. Inherently scalable. Rather than scaling your entire application you can scale your functions automatically and independently with usage.

4. Never pay for idle resources.

5. Built in availability and fault tolerance.

6. Business logic is necessarily modular and conform to minimal shippable unit sizes.

FaaS Disadvantages

1. Decreased transparency. Someone else is managing your infrastructure so it can be tough to understand the entire system.

2. Potentially tough to debug. There are tools that allow remote debugging and some services (i.e. Azure) provide a mirrored local development environment but there is still a need for improved tooling.

3. Auto-scaling of function calls often means auto-scaling of cost. This can make it tough to gauge your business expenses.

4. You now have a ton of functions deployed and it can be tough to keep track of them. This comes down to a need for better tooling (**developmental:** scripts, frameworks, **diagnostic:** step-through debugging, local runtimes, cloud debugging, and **visualization:** user interfaces, analytics, monitoring.

CHAPTER 2 : ARCHITECTURE

2.1 SERVERLESS ARCHITECTURE

Serverless architecture run applications that depend on external Function as a Service and Backend as a Service providers, which run the application code in temporary containers.

Consequently, a serverless architecture includes the following three core components:

API gateway: This is a fully managed service that's used to define, deploy, and maintain APIs. For API integrations, developers use standard HTTPS requests that connect their presentation tier with the functions they write on a serverless platform. Some serverless providers like Amazon offer their own API gateways for their clients.

Function as a Service (FaaS): This is the layer that executes the application code (business logic) through multi-protocol triggers. Function instances are provisioned on request (in a few milliseconds) in compliance with associated configurations.

Backend as a Service (BaaS): This is a cloud computing service model that serves as a cloud-based distributed database, eliminating the need for administrative overhead. BaaS features include cloud storage, push notifications, server code, user and file management, and many other backend services. These services have their own APIs so developers can easily integrate them into their applications.

Hosting a software application on the internet usually involves managing some kind of server infrastructure. Typically this means a virtual or physical server that needs to be managed, as well as the operating system and other web server hosting processes required for your application to run. Using a virtual server from a cloud provider such as Amazon or Microsoft does mean the elimination of the physical hardware concerns, but still requires some level of management of the operating system and the web server software processes.

With a serverless architecture, you focus purely on the individual functions in your application code. Services such as Twilio Functions, AWA Lambda and Microsoft Azure Functions take care of all the physical hardware, virtual machine operating system, and web server software management. You only need to worry about your code.

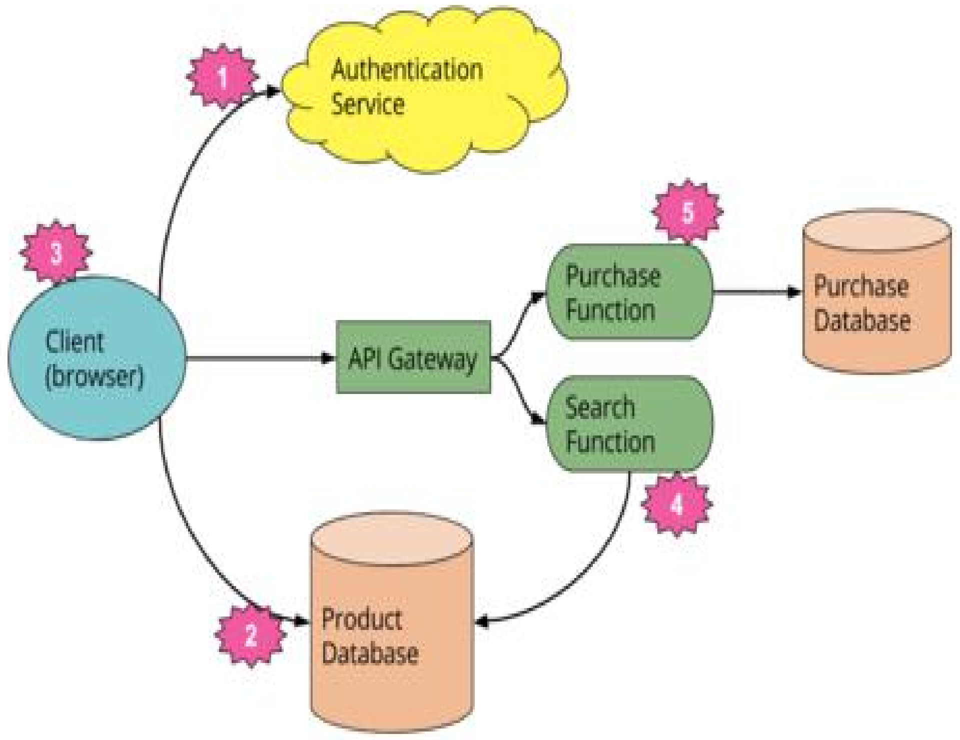

Fig: Serverless Architecture

This is a massively simplified view, but even here we see a number of significant changes:

1. We’ve deleted the authentication logic in the original application and have replaced it with a third-party BaaS service (e.g., Auth0.)

2. Using another example of BaaS, we’ve allowed the client direct access to a subset of our database (for product listings), which itself is fully hosted by a third party (e.g., Google Firebase.) We likely have a different security profile for the client accessing the database in this way than for server resources that access the database.

3. These previous two points imply a very important third: some logic that was in the Pet Store server is now within the client—e.g., keeping track of a user session, understanding the UX structure of the application, reading from a database and translating that into a usable view, etc. The client is well on its way to becoming a Single Page Application.

4. We may want to keep some UX-related functionality in the server, if, for example, it's compute intensive or requires access to significant amounts of data. In our pet store, an example is "search." Instead of having an always-running server, as existed in the original architecture, we can instead implement a FaaS function that responds to HTTP requests via an API gateway (described later). Both the client and the server "search" function read from the same database for product data.

If we choose to use AWS Lambda as our FaaS platform we can port the search code from the original Pet Store server to the new Pet Store Search function without a complete rewrite, since Lambda supports Java and Javascript—our original implementation languages.

5. Finally, we may replace our "purchase" functionality with another separate FaaS function, choosing to keep it on the server side for security reasons, rather than reimplement it in the client. It too is fronted by an API gateway. Breaking up different logical requirements into separately deployed components is a very common approach when using FaaS.

STATE:

FaaS functions have significant restrictions when it comes to local (machine/instance-bound) state. i.e., data that you store in variables in memory, or data that you write to local disk. You do have such storage available, but you have no guarantee that such state is persisted across multiple invocations, and, more strongly, you should not assume that state from one invocation of a function will be available to another invocation of the same function. FaaS functions are therefore often described as stateless, but it's more accurate to say that any state of a FaaS function that is required to be **persistent** needs to be **externalized** outside of the FaaS function instance.

For FaaS functions that are naturally stateless i.e., those that provide a purely functional transformation of their input to their output—this is of no concern. But for others this can have a large impact on application architecture, albeit not a unique one—the "Tweleve-factor app" concept has Precisely the same restriction. Such state-oriented functions will typically make use of a database, a cross-application cache (like Redis), or network file/object store (like S3) to store state across requests, or to provide further input necessary to handle a request.

EXECUTION DURATION

FaaS functions are typically limited in how long each invocation is allowed to run. At present the "timeout" for an AWS Lambda function to respond to an event is at most five minutes, before being terminated. Microsoft Azure and Google Cloud Functions have similar limits.

This means that certain classes of long-lived tasks are not suited to FaaS functions without re-architecture—you may need to create several different coordinated FaaS functions, whereas in a traditional environment you may have one long-duration task performing both coordination and execution.

OPEN SOURCE

So far I've mostly discussed proprietary vendor products and tools. The majority of Serverless applications make use of such services, but there are open-source projects in this world, too.

The most common uses of open source in Serverless are for FaaS tools and frameworks, especially the popular Serverless Framework, which aims to make working with AWS API Gateway and Lambda easier than using the tools provided by AWS. It also provides an amount of cross-vendor tooling abstraction, which some users find valuable. Examples of similar tools include Claudia and Zappa. Another example is Apex, which is particularly interesting since it allows you to develop Lambda functions in languages other than those directly supported by Amazon.

The big vendors themselves aren't getting left behind in the open-source tool party though. AWS's own deployment tool, SAM—the Serverless Applicaton Model—is also open source.

One of the main benefits of proprietary FaaS is not having to be concerned about the underlying compute infrastructure (machines, VMs, even containers). But what if you *want* to be concerned about such things? Perhaps you have some security needs that can't be satisfied by a cloud vendor, or maybe you have a few racks of servers that you've already bought and don't want to throw away. Can open source help in these scenarios, allowing you to run your own "Serverful" FaaS platform?

Yes, and there's been a good amount of activity in this area. One of the initial leaders in open-source FaaS was IBM (with OpenWhisk, now an Apache project) and surprisingly—to me at least!—Microsoft, which open sourced much of its Azure Functions platform. Many other self-hosted FaaS implementations make use of an underlying container platform, frequently Kubernetes, which makes a lot of sense for many reasons. In this arena it's worth exploring projects like Galactic Fog, Fission, and OpenFaaS. This is a large, fast-moving world, and I recommend looking at the work that the Cloud Native Computing Federation (CNCF) Serverless Working Group have done to track it.

2.2 COMPARISON WITH PaaS

Given that Serverless FaaS functions are very similar to Tweleve-Factor Applications, are they just another form of Platform as a service (PaaS) like Heroku? For a brief answer I refer to Adrian Cockcroft.

In other words, most PaaS applications are not geared towards bringing entire applications up and down in response to an event, whereas FaaS platforms do *exactly* this.

If I’m being a good Twelve-Factor app developer, this doesn’t necessarily impact how I program and architect my applications, but it does make a big difference in how I operate them. Since we're all good DevOps-savvy engineers, we're thinking about operations as much as we’re thinking about development, right?

The key operational difference between FaaS and PaaS is *scaling*. Generally with a PaaS you still need to think about how to scale—for example, with Heroku, how many Dynos do you want to run? With a FaaS application this is completely transparent. Even if you set up your PaaS application to auto-scale you won’t be doing this to the level of individual requests (unless you have a very specifically shaped traffic profile), so a FaaS application is much more efficient when it comes to costs.

Given this benefit, why would you still use a PaaS? There are several reasons, but tooling is probably the biggest. Also some people use PaaS platforms like Cloud Foundary to provide a common development experience across a hybrid public and private cloud; at time of writing there isn’t a FaaS equivalent as mature as this.

2.3 COMPARISON WITH CONTAINERS

One of the reasons to use Serverless FaaS is to avoid having to manage application processes at the operating-system level. PaaS services, like Heroku, also provide this capability, and I’ve described above how PaaS is different to Serverless FaaS. Another popular abstraction of processes are containers, with Docker being the most visible example of such a technology. Container hosting systems such as Mesos and Kubernetes, which abstract individual applications from OS-level deployment, are increasingly popular. Even further along this path we see cloud-hosting container platforms like Amazon ECS and EKS, and Google Container Engine which, like Serverless FaaS, let teams avoid having to manage their own server hosts at all. Given the momentum around containers, is it still worth considering Serverless FaaS?

Principally the argument I made for PaaS still holds with containers - for Serverless FaaS **scaling is automatically managed, transparent, and fine grained**, and this is tied in with the automatic resource provisioning and allocation I mentioned earlier. Container platforms have traditionally still needed you to manage the size and shape of your clusters.

I’d also argue that container technology is still not mature and stable, although it is getting ever closer to being so. That’s not to say that Serverless FaaS is mature, of course, but picking which rough edges you’d like is still the order of the day.

It’s also important to mention that self-scaling container clusters are now available within container platforms. Kubernetes has this built in with "Horizontal pod Autoscaling," and services like AWS Fagate also make the promise of “Serverless Containers.”

CHAPTER 3 : METHODS AND TECHNOLOGIES

3.1 RUN APLLICATION IN SERVERLESS ARCHITECTURE:

To create an application, developers should consider how to break it into smaller functions to reach a granular level for serverless compatibility. Each function needs to be written in one of the programming languages supported by the serverless provider.

Function deployment

Deploying a function is considerably different from deploying a traditional system as there's no server on the developer's side. In serverless computing, a developer just uploads function code to the serverless platform and the provider is responsible for its execution as well as for provisioning and allocating resources, maintaining the virtual machine, and managing processes. The platform vendor also takes care of automatic horizontal and elastic scaling.

FaaS functions are executable on a container with few resources. Functions are triggered through requests and events when necessary without requiring the application to be running all the time. Requests that can trigger FaaS functions are specified by the provider. For instance, such triggers can be inbound HTTP requests, scheduled tasks, S3 updates, or messages added to a message bus. Functions can also be invoked via a platform-provided API, externally, or within the cloud environment.

Function limitations

Functions in serverless computing have significant architectural restrictions related to their state and duration of execution. FaaS functions are stateless, which means that they have no machine state. Thus, external databases are necessary to simulate state behavior of functions.

Moreover, functions are limited in their execution time, which determines how long each invocation can be run. This restriction makes developers create several short-running functions instead of one long-running task.

In order to initialize an instance of a function in response to an event, a FaaS platform requires some time for a cold start. This leads to startup latency that depends on various factors and has an unpredictable duration. It may be caused by the programming language, the number of libraries you use, the configuration of the function environment, and so on. Though latency raises many concerns, the latency your app experiences will depend on the traffic and type of your application. If a function repeats frequently, a cold start will be rare, as the platform reuses the instance of a function and its host container from a previous event.

3.2 SERVERLESS PROVIDERS IN THE PUBLIC CLOUD:

The term serverless became popular in 2014 after Amazon launched AWS Lambda, a serverless platform that was followed by the Amazon API Gateway in 2015. By the middle of 2016, the serverless concept had given birth to the Serverless Conference, after which other vendors began to introduce their own solutions to the market.

- ✧ **Amazon's Lambda** is the most mature serverless platform on the market. It supports a variety of programming languages including Java, Python, and Node.js. Lambda takes advantage of most AWS services and allows developers to apply its functions as event handlers as well as to provide glue code when composing services.
- ✧ **IBM Cloud Functions** is another serverless deployment that acts as an event action platform for creating composite functions. It supports Node.js, Java, Swift, Python, and arbitrary binaries embedded in a Docker container. Cloud Function is based on OpenWhisk, which is available on GitHub under an Apache open source license.
- ✧ **Microsoft Azure Functions** lets developers write code for functions like processing images, sending emails, and maintaining files that are executed on schedule or when needed. The platform provides HTTP webhooks, integration with Azure, and support for such programming languages as .NET, Go, Node.js, Python, PHP, and Java. The runtime code is open-source and is available on GitHub under an MIT License.
- ✧ **Google Cloud Functions** is a computing platform that provides basic FaaS functionality to run serverless functions written in Node.js in response to HTTP calls and events from Google Cloud services. It provides development tools such as prediction APIs, translation APIs, and data storage.

In addition to these serverless providers, there are several other serverless projects including OpenLambda, Webtask, LeverOs, Pivotal Cloudy Foundary 2.0, Spontist Functions, and Gestalt Framework.

3.3 AWS LAMBDA FOR SERVERLESS COMPUTING

AWS Lambda is a serverless computing platform implemented on top of Amazon Web Services platforms like EC2 and S3. AWS Lambda encrypts and stores your code in S3. When a function is requested to run, it creates a "container" using your runtime specifications, deploys it to one of the EC2 instances in its compute farm, and executes that function.

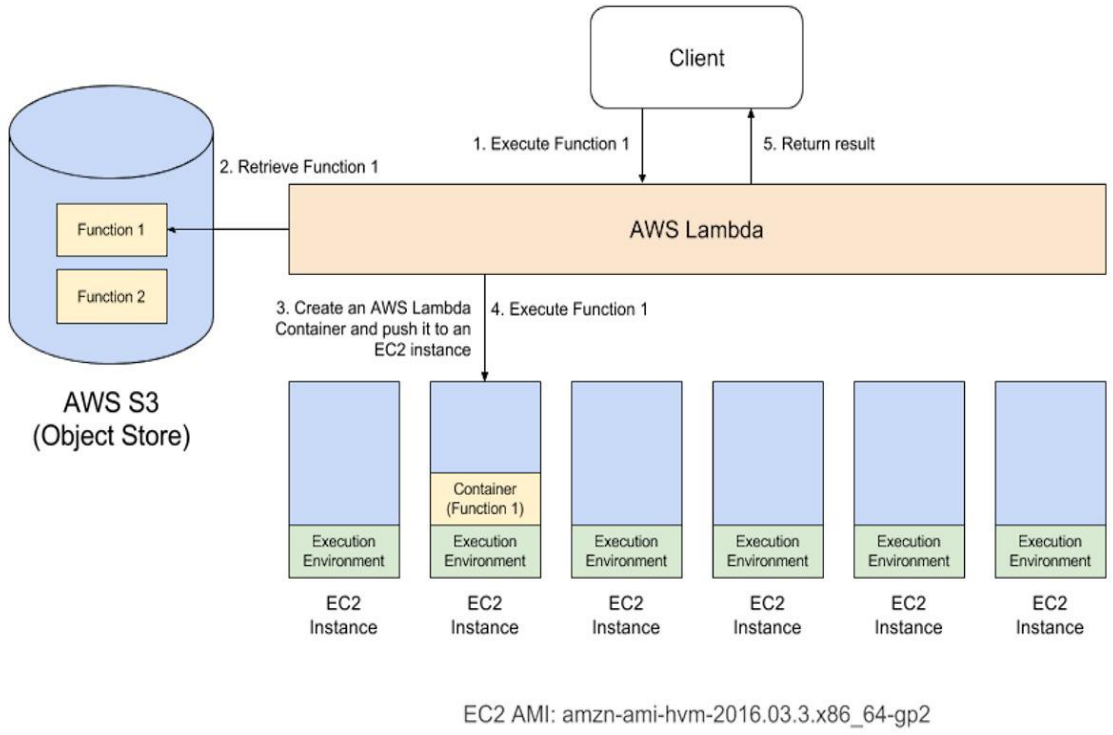

Fig: AWS Lambda for serverless computing

When you create a Lambda function, you configure it in AWS Lambda, specifying things like the runtime environment (we'll use Java 8 for this article), how much memory to allocate to it, identity and access management roles, and the method to execute. AWS Lambda uses your configuration to setup a container and deploy the container to an EC2 instance. It then executes the method that you've specified, in the order of package, class, and method.

At the time of this writing, you can build Lambda functions in Node, Java, Python, and most recently, C#. For the purposes of this article we will use Java.

Lambda Function:

When you write code designed to run in AWS Lambda, you are writing *functions*. The term *functions* comes from functional programming, which originated in lambda calculus. The basic idea is to compose an application as a collection of functions, which are methods that accept arguments, compute a result, and have no unwanted side-effects. Functional programming takes a mathematical approach to writing code that can be proven to be correct. While it's good to keep functional programming in mind when you are writing code for AWS Lambda, all you really need to understand is that the function is a single-method entry-point that accepts an input object and returns an output object.

3.4 NANOSERVICES, SCALABILITY, AND PRICE

Three things really matter about serverless computing: its nanoservice architecture the fact that it's practically infinitely scalable and the pricing model associated with that near infinite scalability. We'll dig into each of those factors.

NANOSERVICES

You've heard of microservices, and you probably know about 12-factor applications, but serverless functions take the paradigm of breaking a component down to its constituent parts to a whole new level. The term "nanoservices" is not an industry recognized term, but the idea is simple: each nanoservice should implement a single action or responsibility. For example, if you wanted to create a widget, the act of creation would be its own nanoservice; if you wanted to retrieve a widget, the act of retrieval would also be a nanoservice; and if you wanted to place an order for a widget, that order would be yet another nanoservice.

A nanoservices architecture allows you to define your application at a very fine-grained level. Similar to test-driven development (which helps you avoid unwanted side-effects by writing your code at the level of individual tests), a nanoservices architecture encourages defining your application in terms of very fine-grained and specific functions. This approach increases clarity about what you're building and reduces unwanted side-effects from new code.

From a design perspective, serverless applications should be very well-defined and clean. From a deployment perspective you will need to manage significantly more deployments, but you will also have the ability to deploy new versions of your functions individually, without impacting other functions.

SCALABILITY

In addition to introducing a new architectural paradigm, serverless computing platforms provide practically infinite scalability. I say "practically" because there is no such thing as truly infinite scalability. For all practical purposes, however, serverless computing providers like Amazon can handle more load than you could possibly throw at them. If you were to manage scaling up your own servers (or cloud-based virtual machines) to meet increased demand, you would need to monitor usage, identify when to start more servers, and add more servers to your cluster at the right time.

PRICING

Finally, the serverless computing pricing model allows you to scale your cloud bill based on usage. When you have light usage, your bill will be low (or nil if you stay in the free range). Of course, your bill will increase with usage, but hopefully you will also have new revenue to support your higher cloud bill. For contrast, if you were to manage your own servers, you would have to pay a base cost to run the minimum number of servers required. As usage increased, you would scale up in increments of entire servers, rather than increments of individual function calls. The serverless computing pricing model is directly proportional to your usage.

CHAPTER 4 : APPLICATIONS

Microsoft, Amazon and Google now got a dedicated branding pages for the topic.

Build virtually any type of application or backend service using a serverless architecture.

✧ Web Applications and Backends

1. Weather Application
2. Mobile backend for social media app

✧ Data Processing

1. Image Thumb nail Creation
2. Analysis of streaming social media data

✧ Building Chatbots

EXAMPLE SCENARIO:

In today's world, there are various use cases for Serverless technologies. Let's take a simple example, imagine you are the CTO of Facebook, one of the key capabilities of Facebook is allowing users to upload their photo's and video's and share it with their friends. Facebook itself is a massive platform, running the platform and scaling it for 2 billion + users will require a huge infrastructure (including many servers). However, this particular functionality can be implemented seamlessly using Serverless model ignoring the entire complexity of the Facebook application.

You can implement the functionality as Amazon Lambda or Azure Function or Google Function, which automatically exposes you an HTTP endpoint, from the front-end web or mobile application you simply upload the content via that endpoint. The data get stored in the relevant backend. This is a simplified example, ideally, you'll add an API gateway and security to the HTTP endpoints, which can also be achieved seamlessly by using either Azure API Management or Amazon API Gateway.

The important point to note here is, this particular functionality needs to scale for 2 billion+ users uploading millions of photos and videos every minute. If Serverless technologies do not exist, this particular capability will take months to implement with huge upfront cost, whereas with Serverless it will be few weeks effort with near zero upfront cost.

CHAPTER 5 : ADVANTAGES AND DISADVANTAGES

5.1 ADVANTAGES

Serverless computing is more beneficial than existing cloud services as it offers better application performance along with reduced operational costs. Let's consider other benefits of cloud computing that attract so much attention to this technology.

- ✧ **Reduced cloud costs** — Serverless computing is based on the principle of pay-as-you-go. It costs nothing when your application doesn't run. Developers pay only for the time when their application executes a user's functions in response to specific events or requests. This model greatly benefits developers, as they can significantly save cloud costs.

- ✧ **Reduced development costs** — With serverless computing, there's no need to handle updates or infrastructure maintenance as developers can now rent most of the resource necessary for software development.

- ✧ **Improved resource management** — Applications developed in serverless environments are more fine-grained. This means that the cloud provider can closely match abstract demand to actual system resources. When an application doesn't run, the cloud provider distributes the server resources among other running applications.Once a triggering event appears, resources are allocated to execute the function.

- ✧ **Elastic scalability** — A serverless architecture has the ability to scale up and down according to application workload. This is achieved by replicating functions. Developers no longer need to purchase additional infrastructure for handling unexpected growth.

- ✧ **Fewer responsibilities for developers** — In the serverless model, the cloud provider has more control over resources and more insight into the context of application behavior. Developers don't need to care about workload intensity, resource distribution, scaling, and application deployment as these issues are in the hands of the cloud provider.

- ✧ **Faster releases and reduced time to market** — To deploy new functions, developers just need to compile their code, zip it, and upload it to the serverless platform. There's no need to write any scripts to deploy functions. This leads to faster releases and time-to-market reductions of up to two-thirds according to a study by Microsoft.

- ✧ **Multi-language support** — Currently, some serverless platforms support multiple programming languages, so developers can choose the most convenient for them.

- ✧ **Agile-friendly development** — Serverless computing allows developers to concentrate on application code rather than infrastructure maintenance. Moreover, it benefits developers through reduced software complexity and better code optimization. For instance, if an application usually takes one second to execute an operation with a hardware server, in a serverless environment it may take only 200 milliseconds, so developers can save 80 percent of their costs.

- ✧ **Built-in logging and monitoring mechanisms** — Serverless providers have developed their own solutions for user logging and monitoring that eliminate the need for developers to purchase third-party tools for similar purposes. In addition, serverless providers offer function-level auditing that ensures the application data privacy that's necessary for full GDPR compliance.

5.2 DISADVANTAGES

- ✧ **Performance:** Infrequently-used serverless code may suffer from greater response latency than code that is continuously running on a dedicated server, virtual machine, or container. This is because, unlike with autoscaling, the cloud provider typically "spins down" the serverless code completely when not in use. This means that if the runtime (for example, the Java runtime) requires a significant amount of time to start up, it will create additional latency.

- ✧ **Resource limits:** Serverless computing is not suited to some computing workloads, such as high-performance computing, because of the resource limits imposed by cloud providers, and also because it would likely be cheaper to bulk-provision the number of servers believed to be required at any given point in time.

- ✧ **Monitoring and debugging:** Diagnosing performance or excessive resource usage problems with serverless code may be more difficult than with traditional server code, because although entire functions can be timed, there is typically no ability to dig into more detail by attaching profilers, debuggers or APM tools. Furthermore, the environment in which the code runs is typically not open source, so its performance characteristics cannot be precisely replicated in a local environment.

- ✧ **Security:** Serverless is sometimes mistakenly considered as more secure than traditional architectures. While this is true to some extent because OS vulnerabilities are taken care of by the cloud provider, the total attack surface is significantly larger as there are many more components to the application compared to traditional architectures and each component is an entry point to the serverless application. Moreover, the security solutions customers used to have to protect their cloud workloads become irrelevant as customers cannot control and install anything on the endpoint and network level such as an intrusion detection/prevention system (IDS/IPS).

- **Privacy:** Many serverless function environments are based on proprietary public cloud environments. Here, some privacy implications have to be considered, such as shared resources and access by external employees. However, serverless computing can also be done on private cloud environment or even on-premises, using for example the kubernetes platform. This gives companies full control over privacy mechanisms, just as with hosting in traditional server setups.

- **Standards:** Serverless computing is covered by International Data Center Authority (IDCA) in their Framework AE360. However, the part related to portability can be an issue when moving business logic from one public cloud to another for which the Docker solution was created. Cloud Native Computing Foundation (CNCF) is also working on developing a specification with Oracle.

- **Vendor lock-in:** Serverless computing is provided as a third-party service. Applications and software that run in the serverless environment are by default locked to a specific cloud vendor. Therefore, serverless can cause multiple issues during migration.

CHAPTER 6 : REFERENCES

1. https://en.wikipedia.org/wiki/Serverless_computing.

2. https://www.infoworld.com/article/3210726/serverless-computing-with-aws-lambda.html.

3. https://geekflare.com/serverless-rising-tech/

4. https://medium.com/@BoweiHan/an-introduction-to-serverless-and-faas-functions-as-a-service-fb5cec0417b2

5. https://techbeacon.com/enterprise-it/essential-guide-serverless-technologies-architectures

6. https://www.apriorit.com/dev-blog/551-serverless-computing