Tampere University

Christopher Allen

MICROSERVICES VS SERVERLESS FUNCTIONS

A comparison of performance and price

Master of Science thesis
Faculty of Information Technology and Communication Sciences TAU
Examiners: Davide Taibi, Nyyti Saarimäki
September 2023

ABSTRACT

Christopher Allen: Microservices vs Serverless functions
Master of Science thesis
Tampere University
Master's Programme in Information Technology
September 2023

Microservices and Serverless computing are two highly popular cloud computing models, with tech giants and cloud vendors employing both models. There exists some crossover between the two in terms of architectural style and purposes, namely concerning event-based applications that are stateless.

In this thesis we compare the two models in terms of performance and cost, to determine if businesses would benefit from migrating from microservices to serverless computing. To measure the cost and performance, an existing microservices demo was deployed on a virtual machine running on Amazon web services' Elastic Compute Cloud(EC2) and then redeveloped as a set of serverless functions running on Amazon web services' Lambda serverless computing platform.

The performance was measured by using a series of mock HTTP requests with different quantities of concurrent mock users (10-1000) and analysing the median response times. In order to measure the costs of the systems, the data from the Amazon billing centre was analysed using a model taking into consideration the number of computing hours used, the cost per serverless function execution, duration etc.

The results showed that despite worse response times at the beginning of the tests, due to a cold start, that the Lambda system performed similar to the EC2 system, and outperforming it at higher levels of concurrent users. It was also concluded that the cost of the EC2 system remains static regardless of the level of concurrent users and requests, but that under a certain amount of requests the lambda system has lower costs.

Keywords: Microservices, Serverless, AWS, Lambda, virtual machines

PREFACE

Serverless is an increasingly popular cloud computing model, which could mark an improvement over other existing cloud models for hosting Microservice applications. The practical work for this thesis was completed in 6 months, the writing in 12.

I would like to express my thanks to my supervisor, Davide Taibi, as well as Xiaozhou Li, Nyyti Saarimäki and Tomas Cerny for the considerable time they have taken to guide and instruct this work.

Finally I want to thank my family and friends, including Fernando Galaz, Kaan Çelikbilek, Veronika Blazhko, and Jani Patrakka.

Tampere, September 2023 Christopher Allen

Tampere, 18th September 2023

Christopher Allen

CONTENTS

1. Introduction 1
2. Background 5
    2.1 Theoretical background 5
        2.1.1 Virtualisation 5
        2.1.2 Docker 5
        2.1.3 Monolithic architecture 6
        2.1.4 Service-Oriented architecture 6
        2.1.5 Microservice architecture 7
        2.1.6 Serverless computing 10
        2.1.7 Cold start 12
        2.1.8 Function-as-a-Service (FaaS) model 12
        2.1.9 Load testing in web applications 12
        2.1.10 Vendor lock-in 14
    2.2 Related works 15
        2.2.1 Serverless testing methods 18
        2.2.2 Economics of Serverless 19
        2.2.3 Architecture Decision on using Microservices or Serverless 20
        2.2.4 Microservice Performance in Container- and Function-as-a-Service Architectures. 21
        2.2.5 Open source serverless frameworks 22
        2.2.6 Serverless utilization in microservice e-learning platform 22
        2.2.7 Serverless Computing: An Investigation of Factors Influencing Microservice Performance 23
3. Study design 24
    3.1 Experiment design 24
    3.2 Data Analysis 29
4. Results 30
    4.1 RQ1. What is the difference in response time? 30
    4.2 RQ2. What is the difference in cost? 32
5. Discussion 35
6. Threats to validity 38
7. Conclusion 40

References 41

LIST OF FIGURES

1.1 A representation of different types of cloud services and what components of the hardware and operating system are under the control of the end-user and which components are entirely controlled by the service provider. . . 2

2.1 A visual representation of a monolithic application architecture. The components that the applications is comprised of are very tightly-coupled and if you remove one of the components then the application cannot function. 7

2.2 A visual representation of a service-oriented architecture (SOA). SOA allows components from different applications on different platforms to communicate with each other over a shared network for enterprise applications. The Enterprise Service Bus facilitates communication between components. This approach can help integrate legacy functionality into new applications with minimal effort. The client on the end can be a human user on a browser or a different service acting as a client requesting a job to be completed. 8

2.3 A concept map detailing a simple eCommerce microservices application split into four different loosely-coupled services and the user interface from which the can be accessed. 10

2.4 Visual representation of a generic serverless architecture using AWS(Amazon Web Services) Lambda as an example. Each serverless function is independent of each other aside from that they use the same API gateway endpoint to receive incoming traffic and then access their own relational database instance. 11

2.5 Visual representation of the scale of three different architectural approaches. The blocks denote how a monolithic structure of tightly-coupled services can be broken down into loosely-coupled services, and then further down into a finite granularity of serverless functions 13

2.6 A concept map detailing the abstraction of a multicloud approach to web application architecture. Different services are hosted by different providers avoiding the pitfalls of being locked into one vendor. 15

2.7 A concept map detailing the abstraction of virtual machines and containers from the underlying infrastructure of a computer system. In this example docker is used. The hypervisor is a layer of abstraction that allows one host system to run multiple guest systems on top of itself and assign resources to them accordingly 16

3.1 Architecture comparison between AWS EC2 (3.1a) and the Lambda serverless 3.1b. The former consists of three loosely-coupled services that are orchestrated using docker containers operating on a docker network that is exposed to external traffic via the EC2 container's public IP (Internet Protocol) address by binding certain internal ports to external ports. The container is monitored using the AWS Cloudwatch tool for debugging purposes. A storage tool is used to upload the relevant files to the container. The latter service consists of three different Lambda serverless functions that are independent of each other but can communicate via direct invocation. These are exposed to external traffic using AWS' API (Application Programming Interface) gateway. 26

4.1 Performance Comparision between AWS EC2 and AWS Lambda in miliseconds (10 Concurrent Users) [47] 31
4.2 Performance Comparison with Different User Numbers. The Y axis denotes the average response time in milliseconds, while the X axis denotes the time interval during the execution of the test that these averages happened at [47]. 31
4.3 Costs Comparison of AWS EC2 and Lambda. The intersection is the point where the expenses of running the Lambda services are as great as running the EC2 service for one month, and beyond it that it becomes more expensive. The cost of the EC2 remains static regardless of the amount of internet traffic that it receives. 33

LIST OF TABLES

2.1 Faas pricing comparison. A table representing the prices and free-tier limits of three different FaaS providers. Amazon Web Services' Lambda, Microsoft's Azure, and Google's Google Cloud Services. The price is given in USD and for each a single invocation of a function is billed depending on how long the function takes to execute in GB-s. [27] 13
2.2 Summary of the mutual differences between Microservices and Serverless functions categorized by duration, dependency, resources, hosting, model, cost, multifunctionality, logging, and size. 15

4.1 Cost Calculation for EC2 Vs Lambda. The measurement of the EC2 usage is the number of hours that the system is online per month, which is estimated from the cost per hour's usage by the number of hours it is used in a day and the number of days it is used per month. 33

LIST OF SYMBOLS AND ABBREVIATIONS

Amazon Web Services	A cloud services vendor
AWS	Amazon Web Services
EC2	Elastic Compute Cloud
Elastic Compute Cloud	A service that provides scalable virtual machines
HTTP	Hypertext Transfer Protocol
lambda	Amazon’s Functions as a service platform
Locust.py	A software tool for creating mock HTTP user requests
Microservices architecture	A software architectural style where the app is comprised of loosely-coupled services
Mongo	A database application.
Monolithic architecture	A software architectural style where the app is comprised of tightly-coupled services forming a monolith
Serverless computing	A software execution model for deploying and executing code without need to manage infrastructure
TAU	Tampereen yliopisto (engl. Tampere University)
TUNI	Tampereen korkeakouluyhteisö (engl. Tampere Universities)
URL	verkkosivun osoite (engl. Uniform Resource Locator)

1. INTRODUCTION

Cloud computing is a paradigm that involves the delivery of on-demand computing resources over the Internet. It enables users to access and utilize a pool of shared computing resources, such as servers, storage, databases, software, and applications, without the need for on-premises infrastructure. The cloud computing model offers scalability, flexibility, cost efficiency, and convenience, making it a popular choice for individuals, businesses, and organizations.

In practice, cloud computing for end users enables many of the most popular web applications today, from Spotify to Netflix, social media platforms, file backups, and code repositories such as GitHub. It allows for users to carry out operations from a web browser that they themselves may lack the necessary hardware to carry out locally.

There are essential characteristics used in defining cloud computing. These characteristics are as follows: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service.

What these characteristics mean is that when using a cloud service the resources should be automatically provided to a consumer without having to go through another human, that the services should be available over network through heterogeneous clients such as computers, and mobile phones. That resources are available to many clients all at once within one shared resource pool and that the resources themselves are being dynamically assigned and reassigned according to the current needs of the clients. The services should be elastic in that they can rapidly be scaled up and down, and finally that the resources be automatically controlled and optimised and that resource usage is available to clients through appropriate levels of abstraction. Cloud computing has been the subject of extensive research, it being one of the most popular research areas in modern computer science. This research tackles many problems in cloud computing such as how to improve existing solutions to make them more efficient, categorizing different cloud computing solutions that are available, how to keep the costs of cloud computing solutions low, to identifying what solution matches what use case.

Cloud computing is modeled based on a 5-layer model: the physical layer, virtual layer, control layer, orchestration layer, and service layer. Different cloud computing solutions are categorized based on how many of these layers are under the control of the user, and

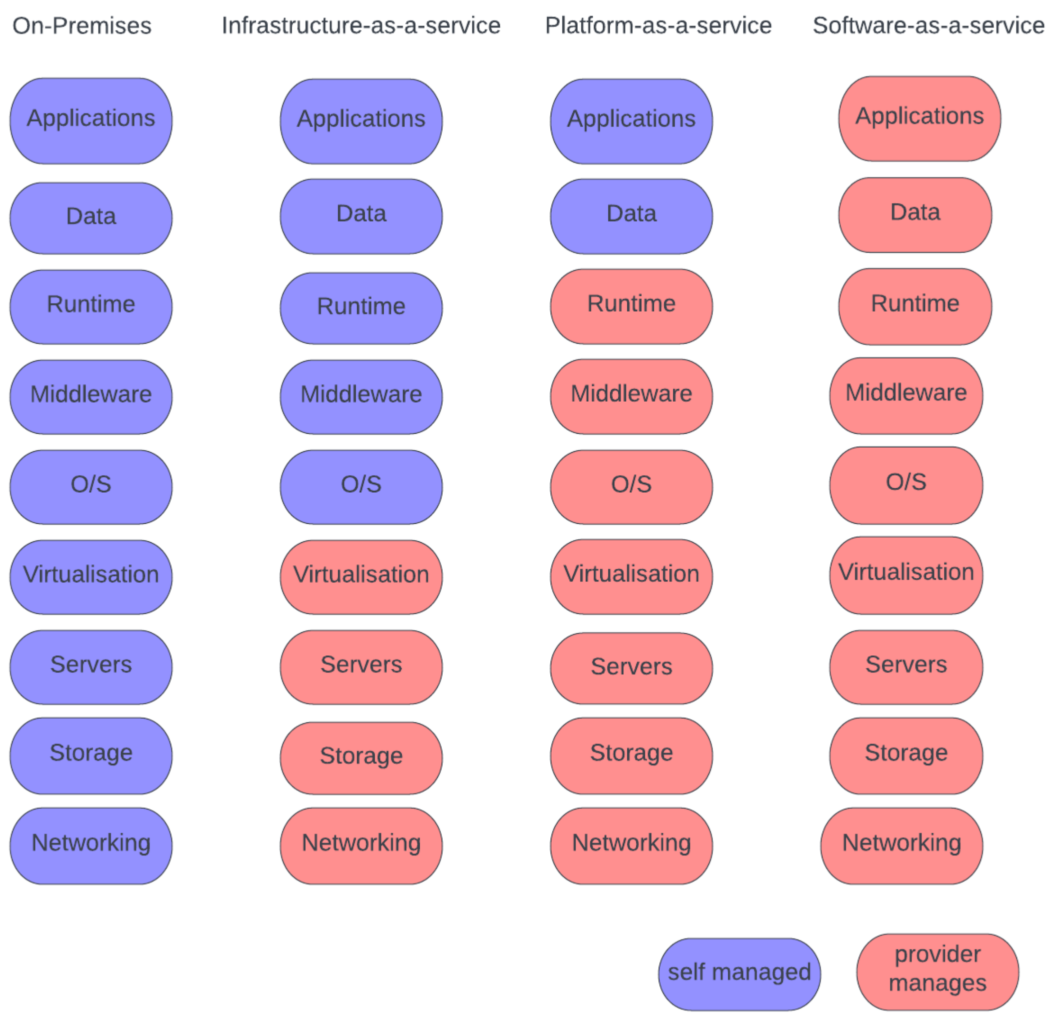


Figure 1.1. *A representation of different types of cloud services and what components of the hardware and operating system are under the control of the end-user and which components are entirely controlled by the service provider.*

how many are completely under the control of the cloud service provider.

There are different models of deployment for cloud infrastructure. These models are differentiated by who owns them, either partly or wholly. The models are as following:

- Private cloud is cloud that is provisioned exclusively for a single organisation and is managed either by them or a third party on or off site.
- Public cloud is provided for the use of the general public by an organisation that operates the system on their premises.
- Community cloud is provisioned for the use of a specific community that can consist of multiple organisations, and may be operated by them or by a third-party with infrastructure existing either on or off premises.
- Hybrid refers to a combination of the above that are bound together by standardised

technology, but remain separate from each other. For example a public cloud that has private services integrated into it.

Microservice architecture is one of the most dominantly popular architectures for cloud-native systems in the industry, where many giant technology companies, e.g., Amazon, LinkedIn, Netflix, and Spotify, are all adopting such an architecture in their systems. In academia, studies on microservice architecture are increasing exponentially in recent years. Many studies focus on the different critical aspects regarding the architectural best practices and issues of microservice architecture, including, the patterns and anti-patterns [1, 2], the decomposition of monolithic systems towards microservices [3], the technical debt of monolithic system migration [4], and so on. Many studies also contribute to the systematic analysis of microservice architecture from different application layers in order to support its monitoring and maintenance, e.g. the architecture reconstruction and visualization techniques [5], static analysis techniques for architectural reconstruction [6], reconstruction visualization [7], and etc.

Compared to microservice, serverless is an emerging technology that enables the reduction of unnecessary overhead for provisioning, scaling, and general infrastructure management [8]. The industry has also seen such benefits and started the migration to the new paradigm of serverless [9]. Many studies also contribute to the theoretical foundation concerning the many aspects of serverless computing, including the patterns and anti-patterns for serverless functions [8], the economic and architectural impact [10], the design and implementation [11], potential issues and solutions [12] and the even broader application of serverless edge computing [13] as well as the platforms for such a purpose [14].

Despite the benefits of serverless mentioned above, many practitioners and companies are still not certain whether it is beneficial to migrate from microservices to serverless or not. Cost and performance are two of the main concerns that have not been addressed in a proper manner. Many studies have contributed to the analysis and modeling of the performance of either microservice [15] or serverless [16], but not in comparison to one another. Several studies also contribute to the comparison of microservice architecture performance and that of monolithic systems [17, 18, 19]. Regarding the cost, studies contributed to the analysis of AWS billing estimation as well as the prediction and optimization of the cost of serverless workflow [20, 21].

This study aims at filling the gap in existing literature for those debating if migration has enough benefits to make it worthwhile, such as potentially lower costs and improved performance with a minimal overhaul to existing code. By comparing these two architectures in terms of cost and performance and determine what benefits might have by migrating if any significant benefits exist at all. To conduct the comparison, we set up an experiment using a demo system. The system is designed with a three-service architecture that is deployed on two different platforms: AWS EC2 for the microservices, and AWS Lambda for the serverless functions.

To evaluate performance, we employ the Locust open-source load testing tool which allows us to simulate different load volumes and measure the response time of both the containerized microservices and serverless functions. By subjecting both configurations to varying levels of load, we can analyze their performance under different conditions and assess their efficiency in handling requests.

As for the cost aspect, We analyze the associated expenses for both containerized microservices and serverless functions. This analysis helps us identify a potential threshold at which the cost comparison between the two approaches may shift. By understanding the relationship between performance and cost, we can provide insights into selecting the most suitable configuration based on specific requirements and constraints.

In addition to the comparison, this study aims at answering the following research questions (RQ):

- ***RQ1.*** *What is the difference in response time between microservice and serverless systems?*
- ***RQ2.*** *What is the difference in cost of running microservice and serverless systems on AWS environments?*

The thesis organized as follows: Chapter 2 provides information about related work on the performance and cost of microservices and serverless architecture, including comparative studies, and some background on serverless testing. Chapter 3 explains the experiment system's design and analysis methods in detail. Chapter 4 presents the results from the experiment. Chapter 5 discusses potential limitations of the work and what could be accomplished in future works. Chapter 6 goes over the potential threats to validity and chapter 7 presents the conclusion to the thesis.

2. BACKGROUND

2.1 Theoretical background

In this Section, we summarize the related works on serverless testing, considering unit, integration, and system-level testing, their advantages as well as disadvantages.

2.1.1 Virtualisation

Virtualisation refers to recreating a virtual instance of a computer system that is abstracted away from the hardware by a hypervisor which is responsibly for the creation and running of virtual machines. There are two distinct types of hypervisors; bare metal and hosted. Bare metal refers to the hypervisor running directly on the computer hardware it is using, and a hosted hypervisor is run as an application on an operating system. Virtual machines are computer systems that run on top of another operating system, and typically have limited access to the host machine's resources. It is important to note that Serverless functions do not fall under the category of virtualisation as the vendor is merely providing access to computing power to run the program on, whereas containers and virtual machines offer greater control over infrastructure.

Containers are similar to virtual machines in the sense that it is an environment isolated from its host system and can run processes independently of it and be exposed to internet traffic. The main difference between the two is that a container is intended only to run a single program and actually shares the same kernel as its host. It only consists of the files and libraries needed to run the application, whereas a virtual machine is a complete computer system and is multi-purpose. [22]

2.1.2 Docker

Docker is a popular tool that is used for creating containers for the purpose of creating platform-independent applications. Docker containers contain files giving it instructions to run the application, what ports to use in the docker network, which to expose to the host network (and potentially internet traffic) and to install all of the libraries and other dependencies that the application may need. Docker can be used to orchestrate multiple applications at once on the same network, run servers and can create data volumes that will persist on the host machine [23].

2.1.3 Monolithic architecture

Monolithic is an older style of software architecture that features very tightly-coupled services, that resemble a monolith. The advantages of this approach to developing applications is that it makes things quite simple given that the entire structure is centralised, and every component shares the same database making the time to deploy applications much faster.

If you remove one component then the entire monolith will come tumbling down, or in other words the application will cease to function. As a consequence of this it is much harder to scale monolithic applications up, and they are a lot less flexible than other architectures. The bigger and more complex the application grows the harder it is to maintain it or to develop it further, and the more data you have the larger the bottleneck in performance. Monolithic architecture is more suited to applications that do not need to be frequently updated or scale to meet user needs.

For an example, Netflix used to be a monolithic application until 2009. The reason for their switching to AWS cloud services was that they experienced a database failure. This was the only component that failed but because of the architecture the entire application went down until the issue was fixed because the other components could not operate independently. Below is a diagram depicting an example of monolithic architecture.

2.1.4 Service-Oriented architecture

SOA (Service-Oriented Architecture) is an architecture that features smaller, more independent, moderately-coupled services compared to those of the monolithic architecture. The time to develop and implement changes is much faster as the application can still function if one component is removed. SOA is able to take and re-purpose existing legacy functionalities by exposing it via interface to newer web applications when before these functions might have been locked into a monolithic architecture .

SOA makes use of a ESB (Enterprise Service Bus), which allows different components of different applications across different platforms communicate with each other and allows components to be reused without much extra effort to integrate them. SOA therefore is used for business applications in enterprises. When the client uses a service it is abstracted away from them. All they know is the information about what the service does, how to communicate with it, and any documentation that has been provided on the service registry that lists all of the available services on the network.

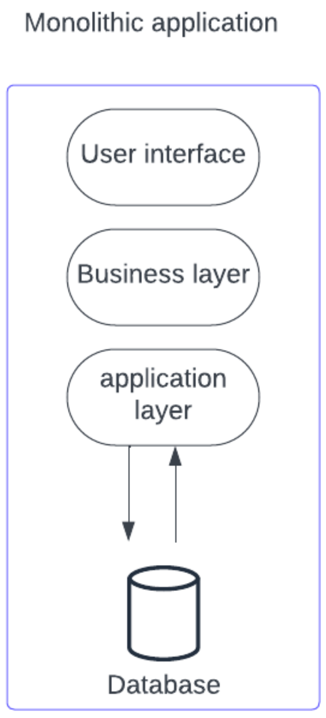


Figure 2.1. *A visual representation of a monolithic application architecture. The components that the applications is comprised of are very tightly-coupled and if you remove one of the components then the application cannot function.*

An example of SOA would be in a hospital environment. There are many different departments in a hospital that serve different functions but have overlap, especially when it comes to the supporting systems that they need access to. For example, a pharmacist and a doctor both need to be able to access a patient's history and record changes in prescribed medication. Instead of redeveloping the same system for both departments which would require a lot of effort, they can instead access an organisation-wide patient database from the client in their department.

2.1.5 Microservice architecture

Microservices refer to lightweight web application modules that are loosely coupled together. Microservices evolved from Monoltihc and Service-oriented architectures. These services that comprised these applications were tightly coupled and structured in such a way that if you removed one service or component of the application, then the application could not function until you replaced it with the same/different component, making it difficult to push any kind of changes without taking the whole system offline in case of unforeseen problems. [24]

With microservices, any service at any granularity can be exposed, and the architecture

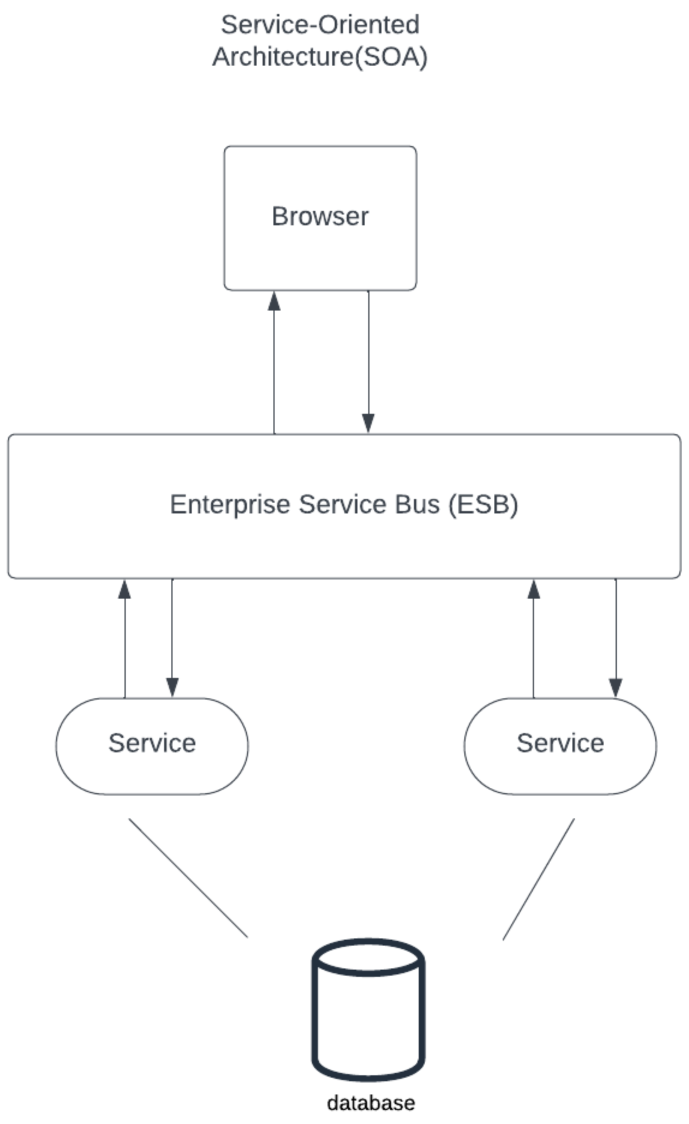


Figure 2.2. *A visual representation of a service-oriented architecture (SOA). SOA allows components from different applications on different platforms to communicate with each other over a shared network for enterprise applications. The Enterprise Service Bus facilitates communication between components. This approach can help integrate legacy functionality into new applications with minimal effort. The client on the end can be a human user on a browser or a different service acting as a client requesting a job to be completed.*

style is very flexible and more simplistic, especially the maintenance of microservices as if one of these services is removed then the rest of the services can still function in a reduced capacity until the other services are brought back online or is replaced. This also lends itself well to continuous deployment, as the size of the application won't grow the same way a monolithic application does, and it is easier to push updates to different modules. To give an example of a microservices structure, if we take an online shopping website and look into the backend of it, we can see that it's split into separate modules:

- The ***search engine*** service would handle the functionality of searching and retrieving products based on user queries. It would have its own codebase, data storage, and API endpoints dedicated to this task.
- The ***item database service*** would manage the storage and retrieval of product information, such as names, descriptions, prices, and inventory levels. It would handle tasks related to product management, updates, and data synchronization.
- The ***shopping cart service*** would take care of managing the user's selected items, allowing users to add, remove, and update their cart contents. It would handle cart-related operations, such as calculating totals, applying discounts, and saving cart information for future sessions.
- The ***payment service*** would handle secure payment processing, integrate with external payment gateways, and manage transactions. It would ensure that users' payment information is handled securely and that orders are processed accurately.

By splitting these functionalities, the online shopping website gains several advantages as each microservice can be developed, deployed, and scaled independently, allowing for faster development cycles and easier maintenance. It also enables teams to work on different services concurrently, fostering parallel development and reducing dependencies. The modular nature of microservices allows for better fault isolation—If one service experiences an issue or requires an update, it can be addressed without affecting the rest of the system.

The costs of microservices depend on how they are hosted. If they are hosted using on-site servers then the costs largely come from maintenance of the servers, licenses for software, and maintaining the production pipeline. In scenarios where the developer is not responsible for the hardware, these costs are streamlined and become about paying for the service from a cloud vendor used to host the application, which can reduce costs. These solutions scale depending on how much storage, memory, and processing power the application requires, they can also be used in conjunction with other services such as hosted databases.

The drawback to such an approach is that there is increased operational complexity and overhead. Issues need to be considered such as balancing the load between services,

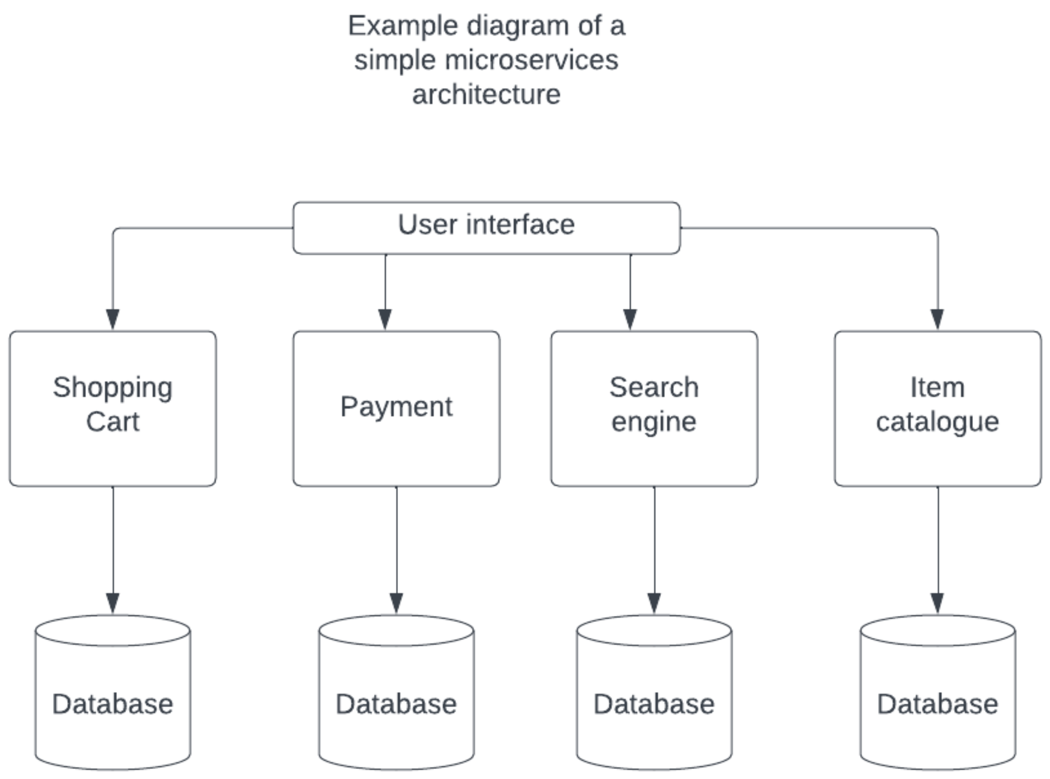


Figure 2.3. *A concept map detailing a simple eCommerce microservices application split into four different loosely-coupled services and the user interface from which the can be accessed.*

the complexity of interactions between services, service discovery, and fault tolerance. These can be mitigated with careful planning of the individual services

2.1.6 Serverless computing

Serverless computing is a cloud computing model where developers can build and run applications without the need to manage or provision servers. In a traditional server-based architecture, developers are responsible for setting up and maintaining servers to host their applications. However, in serverless computing, the infrastructure management is abstracted away, allowing developers to focus on writing code and building applications [25].

In a serverless architecture, the cloud provider takes care of all the server management tasks, such as server provisioning, scaling, and maintenance. Developers only need to write and deploy their code in the form of functions or small units of logic, often referred to as serverless functions or function-as-a-service (FaaS). These functions are event-driven, meaning they are triggered by specific events such as HTTP requests, database updates, or scheduled events.

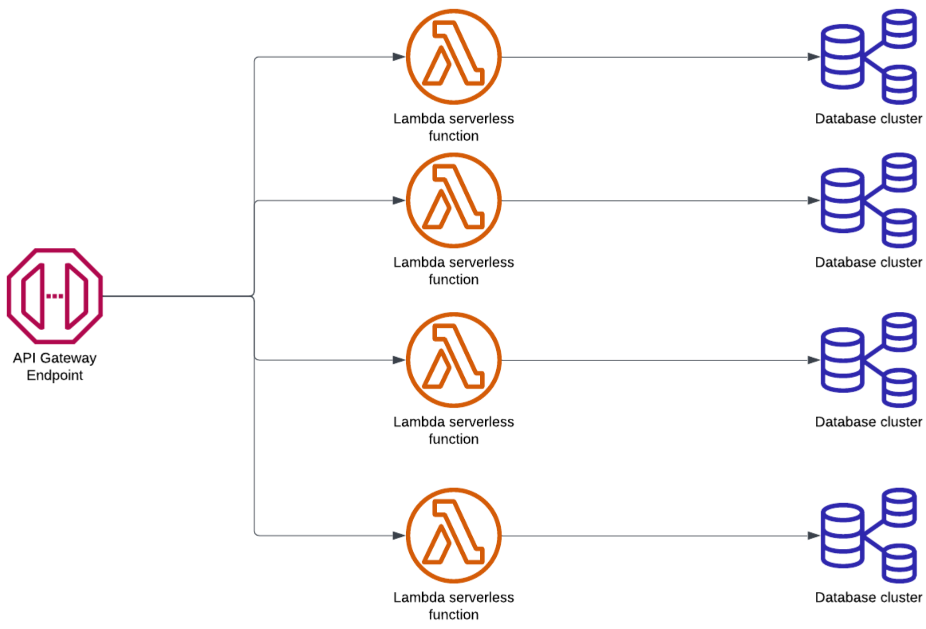

Lambda serverless function
Database cluster
Lambda serverless function
Database cluster
API Gateway Endpoint
Lambda serverless function
Database cluster
Lambda serverless function
Database cluster

Figure 2.4. *Visual representation of a generic serverless architecture using AWS(Amazon Web Services) Lambda as an example. Each serverless function is independent of each other aside from that they use the same API gateway endpoint to receive incoming traffic and then access their own relational database instance.*

When an event occurs, the cloud provider automatically allocates the necessary resources and executes the corresponding function. This on-demand scaling ensures that applications can handle varying workloads efficiently, without developers needing to worry about infrastructure scalability.

Serverless computing offers several benefits:

1. They provide developers with a highly scalable and elastic environment. Functions can scale automatically based on the incoming workload, ensuring optimal performance and resource utilization. This scalability is achieved by spinning up new instances of functions as needed and shutting them down when they are no longer required.
2. Allows for cost optimization. Users are billed based on the actual execution time and resource consumption of their functions, rather than paying for idle server time. This makes it cost-effective, especially for applications with sporadic or unpredictable workloads.
3. Inherent high availability and fault tolerance. Cloud providers handle the underlying infrastructure, ensuring that functions are automatically replicated and distributed across multiple data centers. This redundancy helps mitigate the risk of single-point failures and ensures that applications remain highly available.

2.1.7 Cold start

Cold start is a phenomenon that occurs in server computing where an inactive serverless function is pinged and the cloud service needs to allocate resources to deal with the incoming request(s). This causes a significant delay in the response time of the function and increases it from a matter of milliseconds to seconds. There are multiple ways to avoid this issue, for example by pinging the function to keep it active. [26]

2.1.8 Function-as-a-Service (FaaS) model

The Function-as-a-Service (FaaS) model is a form of serverless computing that is provided as a service to cloud users. In this model, users provide functions to the cloud provider, who takes care of the entire operational lifecycle, from deploying the function to ensuring security patches are applied. FaaS offers a high-level abstraction of distributed computing elements, reducing the need for users to be experts in distributed systems and letting them focus on business concerns. Major public cloud providers already offer FaaS solutions, and numerous FaaS platforms and serverless projects are being developed in the open-source community. The potential of serverless computing has also triggered our academic interest.

The serverless market has an ever increasing number of shareholders. Among these are tech giants Amazon, Google, Microsoft with AWS Lambda, Google Cloud Platform, and Microsoft Azure. Serverless providers have a level of abstraction that obscures the inner workings of the serverless runtime environment from the users, they cannot see or interact with the underlying infrastructure, or the resources. Serverless systems are suited for use with lightweight event-driven functions, given that the resources are only provisioned during the execution time, the serverless application must be stateless.

The cost of serverless services is based on usage. The more traffic your application receives, the more times the serverless functions are pinged adding to usage. Some services like Lambda give users X number of free computing time/usage per month and then charging after that has been exceeded. This contrasts with a service that provides a virtual machine that is always on, where the cost is constantly accumulating while it is running, even if there is no traffic. Serverless systems have some drawbacks such as long response times from a cold start, the apps will need to be stateless and event-driven. There is also the potential for vendor lock-in.

2.1.9 Load testing in web applications

Load testing in web applications refers to a method of testing, where the system in question is subjected to different loads, and determine how much of a load of end user traffic

Table 2.1. *Faas pricing comparison. A table representing the prices and free-tier limits of three different FaaS providers. Amazon Web Services' Lambda, Microsoft's Azure, and Google's Google Cloud Services. The price is given in USD and for each a single invocation of a function is billed depending on how long the function takes to execute in GB-s. [27]*

	free tier	pricing
AWS Lambda	1 million/month 400,00GB-s	0.0001667 per GB-s
Microsoft Azure	1 million/month 400,000GB-s	0.000016 per GB-s
Google Cloud Functions	2 million/month 400,000GB-s	0.0000004 per GB-s separate billing for memory and CPU
IBM	1 million/month 400,000GB-s	0.000017 per GB-s

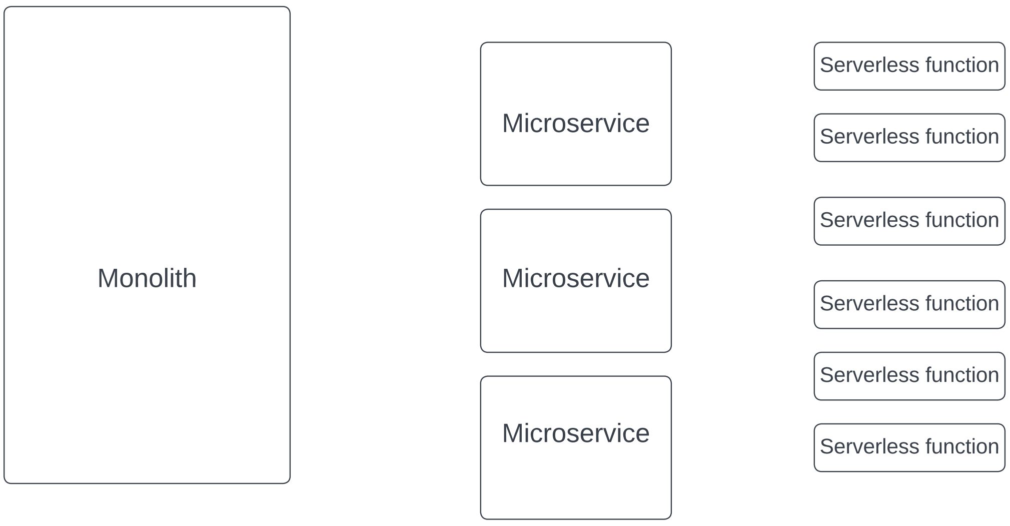


Figure 2.5. *Visual representation of the scale of three different architectural approaches. The blocks denote how a monolithic structure of tightly-coupled services can be broken down into loosely-coupled services, and then further down into a finite granularity of serverless functions*

the system is able to handle. An important metric in load testing is the time it takes for the client to get a response from the system from the moment that the request is sent. Another important metric to look at is the number of concurrent users and the number of hours they log as the performance of the system is directly impacted by the number of user sessions it has to handle. By taking the total time that the test was performed for and the number of concurrent users we can calculate the "throughout" of the application which is the number of user requests per second that have been processed for the test's duration.

2.1.10 Vendor lock-in

Vendor lock-in is a phenomenon that occurs when a company is too dependent on one particular vendor for the services/environment which they use to develop their web application and use only that one vendor's services for everything. The issue with this is that it can have the potential to severely limit the performance of the application and cost more money in operations. It also poses a problem in the future as it can prevent the application being migrated away from the vendor in question. Some of the reasons that we may want to migrate partially or entirely from a particular vendor is the potential for a complete security breach of the application, as if one of the services the application uses is compromised then that means that the other services can be too. The quality of the service might deteriorate over time and it may become outdated due to a lack of updates to the services on the vendor's part. A vendor may also decide to cut support for a particular framework that the application is dependent on. Finally the vendor may cease to exist or experience a long period of downtime and an interruption of services. One way to avoid vendor lock-in is to use a multicloud approach where different services from different vendors are used in conjunction.

Multicloud

Multicloud refers to an approach to serverless computing where different serverless components from different cloud vendors are used in conjunction with one another to improve performance by mitigating the drawbacks of each individual service. It could potentially be cheaper than using one service as well because workloads are distributed across multiple vendors the execution times and resources used are lower than if one vendor was used to handle every incoming workload.

Microservices and serverless functions are not mutually exclusive and could be used in conjunction with each other to mitigate some of the issues to adopting either of them. The mutual differences between these functions are described in Table 2.2.

When we measure the performance of both microservices and serverless functions we

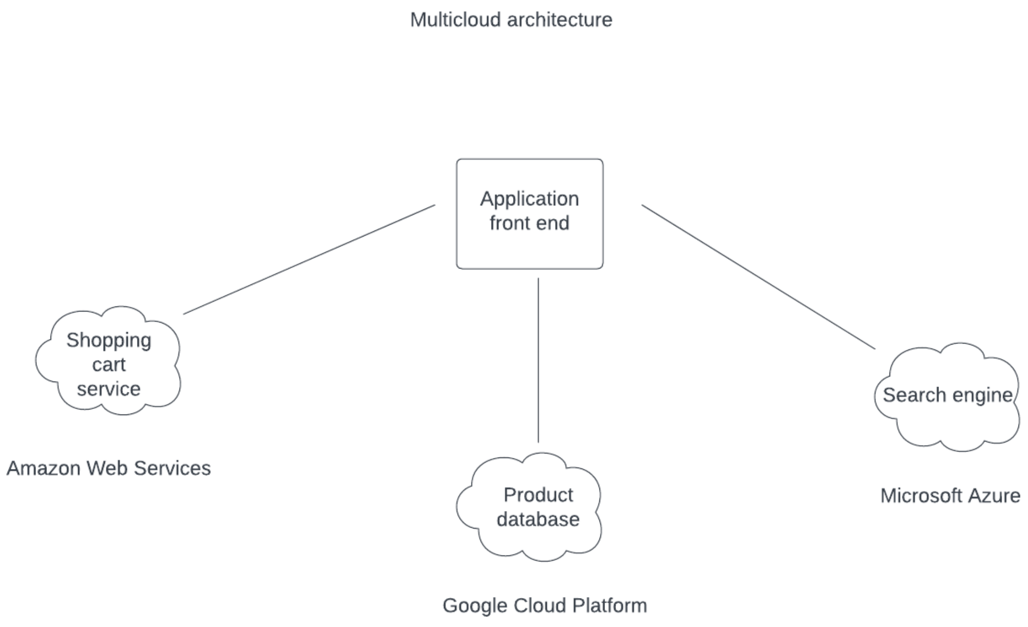


Figure 2.6. *A concept map detailing the abstraction of a multicloud approach to web application architecture. Different services are hosted by different providers avoiding the pitfalls of being locked into one vendor.*

Table 2.2. *Summary of the mutual differences between Microservices and Serverless functions categorized by duration, dependency, resources, hosting, model, cost, multifunctionality, logging, and size.*

	Microservices	Serverless functions
Duration	Continuous	Execution-based
Dependency	Loosely-coupled	Singular
Resources	Always available	Dynamically-allocated
Hosting	Self/service-based	Service-based
Model	Service-oriented/cloud computing	Cloud computing
Cost	Continuous + execution based	Execution-based
Multifunctional	Yes	No
Logging	Self-managed	Service
Size	Small-medium	Small

measure the response time for a single request to complete, relative to the amount of internet traffic that is being processed during that time frame. This is essential in web applications as a long delay in response times can have a large impact on not just one, but potentially hundreds of users as computing resources are tied up as new requests are made.

2.2 Related works

Performance engineering for microservices, in terms of testing, monitoring, and modeling, is one of the challenges in the related domain [28]. Especially regarding perfor-

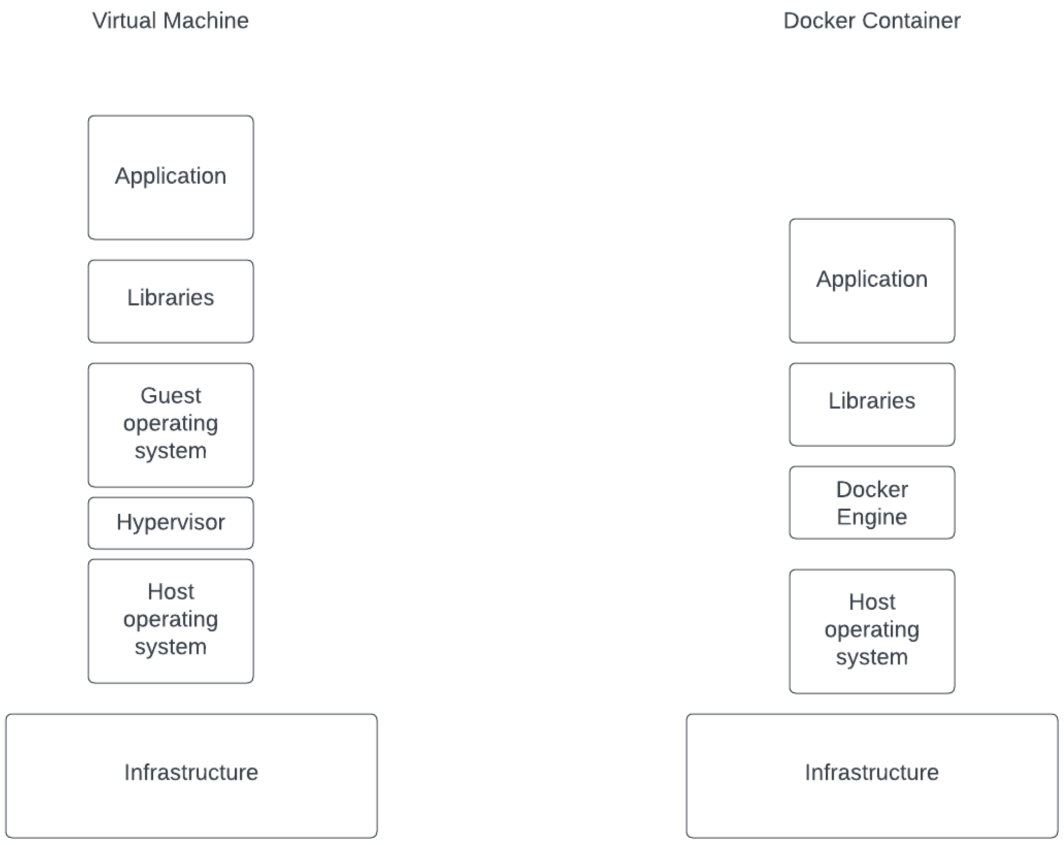


Figure 2.7. *A concept map detailing the abstraction of virtual machines and containers from the underlying infrastructure of a computer system. In this example docker is used. The hypervisor is a layer of abstraction that allows one host system to run multiple guest systems on top of itself and assign resources to them accordingly*

mance testing, many approaches have been proposed. For example, De Camargo et al. [29] proposed an automatic testing method where each microservice shall provide a test specification. Regarding the performance comparison between microservices and monolithic systems, Auer et al. [30] proposed an assessment framework that encompasses the measures of function suitability, performance efficiency, reliability, maintainability, process-related, and cost as the key dimensions. Regarding the performance comparison between microservice and monolithic systems, Blinowski et al. [19] conducted a series of controlled experiments in different deployment environments to verify the different benefits of the migration from monolithic systems to microservice in various context settings. Gos and Zabierowski [18] also conducted experiments comparing the performance of microservice and monolithic systems in terms of response time for different request numbers and indicating the pros and cons of both architectures. Al-Debagy and Martinek [17] drove a comparison experiment on load testing and service discovery scenarios with specific configurations between the two architectures.

On the other hand, many studies also contributed to the performance engineering regarding serverless applications in terms of the various quality aspects of the architecture. Eismann et al. [31] conducted a case study towards investigating the stability of performance tests for serverless applications by comparing the results with different load levels and memory sizes. Their findings show improvement in the response time (faster responses) with higher workloads and also larger function sizes, as well as performance fluctuations in the short-term and long-term within the observation period. Lloyd et al. [9] investigate the influencing factors of infrastructure elasticity, load balancing, provisioning variation, infrastructure retention, and memory reservation size by comparing such attributes of AWS Lambda and Azure Functions. Their results indicate that extra infrastructure is provisioned to compensate for the initialization overhead of COLD service requests. Lee et al. [32] compared and evaluated concurrent invocations on Amazon Lambda, Microsoft Azure Functions, Google Cloud Functions, and IBM Cloud Functions. Their results show that the elasticity of Amazon Lambda outperforms the others regarding throughput, CPU performance, network bandwidth, and file I/O in terms of concurrent function invocations for dynamic workloads. Yu et al. [33] proposed a benchmark suite for characterizing serverless platforms and compared the evaluation results on AWS Lambda, Apache OpenWhisk, and Fn serverless platforms.

Despite the studies in performance engineering for either microservice or serverless systems, limited contributions to the performance comparison between them. Fan et al. [20] performed a performance comparison study of a cloud-native application regarding its reliability, scalability, cost, and latency between microservices and serverless strategies. They conducted experiments using an employee time-sheet management system developed with Node.js with three main modules when their deployment strategy is 5 containers for 5 main cases for microservices and 6 Lambda functions for serverless. The results show that serverless suffers from cold-start issues and is outperformed by microservice with small size and repetitive requests. Their results also show that microservices suffer from the load balancing and traffic redistribution problem, nevertheless, they do not provide performance comparison with different request load numbers.

A case study is described where this approach was applied to a real-world cloud-native monolithic application. Two parts of the application that were suitable for migration - a reporting service and a data processing service - were identified and migrated into microservices and FaaS while leaving the rest of the application in the monolithic architecture.

The benefits of this approach are highlighted, including reduced risk, reduced downtime, and improved maintainability. It is also noted that the partial migration approach allows developers to gradually adopt microservices and FaaS without disrupting the existing application.

The challenges faced during the migration process are also discussed, including the need for careful planning and coordination, as well as the need to maintain compatibility with the existing monolithic architecture. Regarding the comparison of costs between approaches, a study conducted by Fan et al. [20] compared the costs of serverless and microservice systems based on the number of requests and the code execution time. While this method is useful, it does not account for the concurrent load as an influencing factor on cost.

Additionally, there have been several other studies that have investigated the cost of serverless and microservice systems. For example, Eismann et al. [21] proposed a method to predict the cost of serverless workflows, which allows for the prediction and optimization of the expected cost of a serverless workflow. This type of analysis can help developers make informed decisions about the most cost-effective way to implement their applications.

Mahajan et al. [34] explore the optimal pricing for serverless computing, the work discusses the modeling of cloud computing services where a client can use both serverless computing and virtual machines simultaneously to split jobs for the same web application. The cost model for these services is based on fixed unit time prices for both types of services, but the cost of serverless computing is proportional to the sum of the running times of the jobs processed. The client's load is modeled as a continuous arrival of jobs split randomly between the rented resources, and each rented resource is modeled as an M/G/1 queue (for VMs) or an $M/G/\infty$ queue (for serverless computing). The client has an explicit performance constraint, such as an upper bound on the average response time of jobs, and this constraint is used to determine the number of VMs rented and the amount of load assigned to serverless computing to minimize cost.

Furthermore, they investigated the economic benefits and performance tradeoffs when using virtual machines and serverless computing simultaneously. The results indicate that if there is a load that if using a serverless function for executions is cheaper than the renting of a virtual machine then the serverless option will always be the more cost effective. This applies to scenarios where the cost of using serverless for a load equals that of a virtual machine and the load is more than what a virtual machine can handle.

2.2.1 Serverless testing methods

Different traditional testing methods can be used in the testing of Serverless computing, despite the user typically only having access to the environment in which the code is executed and not the underlying infrastructure. These methods include: Unit testing, hybrid Testing, cloud-integration testing, load testing, and end-to-end.

Regarding Unit Tests, Zambrano [35] employs local versions of services, e.g. databases, for testing. There are ways to simulate the necessary cloud stack in a local environment, with AWS providing a local executable for DynamoDB. Certain third-party solutions like

LocalStack can simulate a broader range of AWS services locally using Docker containers.

In serverless Hybrid testing, integration tests are conducted using real/simulated cloud resources, however the tests are executed from the local machine. Tools such as Locust.py can be used for this purpose. If we want to test the serverless functions in an environment that is unavailable to public internet traffic we can .

In integration testing, the serverless function is deployed in the cloud and tested against real-world services, [36, 37, 38, 39, 40] there are numerous sources that vouch for the use of regular cloud services as they provide more reliable results in terms of integration functionality during real-world scenarios. Local simulations cannot provide a precise one-to-one recreation of their cloud counterparts and the factors affecting their performance [41]. Hence, cloud integration testing is the preferred choice for validating integration with cloud services. However, testing deployed functions in the cloud often results in black-box testing since there is no access to the application runtime, making it difficult to observe code execution or inject mocks. In cases where specific integrations need to be mocked, hybrid testing is likely the more suitable option [39].

Serverless functions can be directly invoked through multiple tools, such as AWS CLI or directly from the website. Invocations can also occur through indirect triggers such as an API Gateway, and these triggers can be used through tools such as curl or Postman. Lambda can also be simulated in a local environment.

The System Testing approaches outlined in the literature primarily focus on load testing and end-to-end testing. In end-to-end testing, the goal is to invoke the application by simulating a real-world execution scenario. There are various ways to perform end-to-end testing for a cloud system, depending on the use case and method for invocation. In a scenario where the trigger is an external action from another service, a test case could involve performing said action and then monitoring the Lambda function for its execution. However, this approach can be challenging to implement, prone to errors, and typically results in longer test duration, often relying on timers [42].

2.2.2 Economics of Serverless

A study was carried out to find out what assumptions needed to be made to compare the pricing strategies of AWS Lambda, and AWS EC2 [43]. The instances that were made was that:

1. The code worked seamlessly on both instances
2. The application is auto-scalable
3. Saving in IaaS administration costs were not a factor.

4. A 1:1 throughput ratio is used meaning that the amount of memory used by both should be roughly the same for a single request, for as much memory as is available to the EC2 instance.

In this work, a formula is given for working out the cost of running a lambda instance for a month, which is calculated based on the number of requests made, the memory allocated, and the (expected) execution time. This contrasts with the factors making up the formula for calculating the cost for an EC2 instance. In an EC2 instance, the factors include; the time period, the maximum number of requests per second that can be handled by the instance, the number of requests per second, and the cost per time unit.

The comparison between the costs of running these two instances based on the formulas found that until a certain point, Lambda was cheaper than the EC2 instance, but after that point the cost of lambda rose more sharply, leading to the conclusion that the Lambda instance was the better option if the expected traffic was low enough, with the system experiencing periods of inactivity. This point, where the cost of lambda passes that of the m4.4xlarge EC2 instance can be higher or lower depending on the application being run. Across all of the cases, the cost-to-request ratio of the m4.4xlarge remains consistent [43].

2.2.3 Architecture Decision on using Microservices or Serverless

Functions with Containers

In this article, cloud technologies are discussed in detail and analyzed in order to compare the advantages and challenges associated with each and then suggest a solution to move forward with respect to the cloud.

The systems being compared are virtual machines, containers, and a serverless application. The issues with serverless that are addressed are the vendor lock-in, monitoring the application, and testing the application can be more difficult with integration testing because the units of integration are smaller than in other application architectures.

Docker containers simplify infrastructure management from virtual machines and are also more resource efficient. By abstracting the infrastructure entirely so that the user only needs to upload code to run, serverless manages to be the simplest one of the three to run and maintain. The article notes that Docker is not mutually exclusive with Serverless and that Docker can be considered Serverless in the sense that functions can be containerized, deployed, and used in the same way that Serverless functions are. The advantage to this is that there are some tasks that can't be accomplished with Serverless functions that could instead be accomplished with containers running functions and vice versa.

The main finding from the article was that Serverless requires a cloud service provider whether they are operating in a container, or as serverless functions. The approach of running microservices without serverless computing works better when it is expected that the code will always be running instead of only needing to take resources at execution time. The article then recommends that the ideal option is for microservices to be run in containers, although splitting microservices into separate functions impact the performance and the ability to monitor/measure the microservice.

2.2.4 Microservice Performance in Container- and Function-as-a-Service Architectures

This paper [44] proposes an evaluation model of the functions-as-a-service model, that is implemented as an open-source platform on a private cloud environment, and then compared against a container-as-a-service. The testing architecture is comprised of docker for the container approach, Apache OpenWhisk for serverless architecture, and Prometheus to collect the relevant data.

The metrics analyzed here are response time, and CPU and memory utilization. Two microservices are deployed to be tested; a face recognition service, and an image conversion service. The language used in the tests is Python. The tests are carried out on two virtual machines on the same server that both have 4 threads and 4GBs of RAM each, and either OpenWhisk or Docker installed as per the approach that that machine simulates [44].

Two different approaches were taken to testing, interval and sequential. In the first case 1000 requests were sent 60 seconds apart, and in the second 3000 were sent without concurrency immediately after the previous request was resolved. By utilizing both approaches the "cold start" nature of the serverless approach can be properly evaluated.

Evaluating the response time, the Docker approach had an average of 2.3 seconds and 1.1 seconds for each microservice. The OpenWhisk approach had an average of 4.8 and 3.6 from a cold start for the services, however, this dropped drastically to 0.7 and 0.2 seconds from a warm start making it faster than the Docker container. The results for the CPU utilization returned that the Docker container utilized 34 percent of CPU resources while OpenWhisk only consumed 18.

RAM utilization is poorer in the serverless approach, with 1.9GB of RAM used while the Docker container only uses 900MB. The reason for this and for the difference in CPU utilization is because due to the architecture of OpenWhisk, actions are kept in active memory which saves the script needing to be interpreted again, sacrificing RAM for lower CPU utilization.

The findings from this paper show that there is considerable overhead in a cold-start of a FaaS system that makes it perform slower than the CaaS, however, this becomes the opposite in the case of a sequential series of requests that prevents a shutdown and keeps the system warmed up.

2.2.5 Open source serverless frameworks

As mentioned earlier in this chapter, vendor lock-in is a problem with serverless frameworks, and cloud services in general. Diversifying the vendors that a company uses is a method to prevent this lock-in and protect the application from severe breaches or other potential issues by making migration easier.

However, frameworks such as a AWS Lambda or Microsoft Azure have constraints placed on them in terms of storage, duration and concurrency. One way to combat this is by considering open source frameworks that can reduce costs and further mitigate the issue of vendor lock-in. Mohnaty et al. [45] evaluated a number of open source frameworks by orchestrating them simultaneously with Kubernetes and then measuring and comparing the response times with different numbers of concurrent users, and with different numbers of replicate functions (1,25,50). The selected frameworks were Kubeless, Fission, and OpenFaaS. To reduce overhead in function logic and dependencies, a simple function was deployed that takes a string as input and returns that same string. Kubernetes cluster was selected for the orchestration of the tests as it was the only orchestrator that is supported by all three of the frameworks being tested.

The performance of the three frameworks was evaluated by deploying them on the Google Kubernetes Engine, with a cluster consisting of three identical worker nodes and the defualt settings enabled for each framework. To generate the requests, Apache Benchmark was deployed on a virtual machine and sent 10,000 requests at different levels of concurrency [45].

2.2.6 Serverless utilization in microservice e-learning platform

In this paper a hybrid system of microservices and serverless functions deployed via AWS EC2 and Lambda was proposed, and a comparison of the performance of the two carried out by deploying an application that has two backends, one serverless and one based on microservices. Both of the backends have the same methods, but the serverless backend is only called upon during low traffic periods and starts 30 minutes before the microservice backend becomes inactive, the reverse happens when the app enters high traffic hours [46]. All of this is controlled from the frontend.

Apache Jmeter was used to evaluate the performance and scalability of both of these systems. The tests consisted of a ramp test starting from 50 concurrent users and adding 100 users every two minutes of the test, and then a spike test of 200 concurrent users that spikes to 400 and then 600 concurrent users.

The results showed that the EC2 system had better average response times using one instance up until 500 concurrent users. There was a significant spike in the result times of the lambda system from 300 concurrent users onward. As for the spike tests, there was no noticeable impact on the performance of the EC2 system, however the Lambda system experienced a massive increase in response time despite being stable during the first spike.

2.2.7 Serverless Computing: An Investigation of Factors Influencing Microservice Performance

Another study that provides some methodology and a benchmark for our own results is "Serverless Computing: An Investigation of factors influencing Microservices Performance.". This study tackles the issues regarding the performance of serverless systems such as the cold start problem and compares microservices deployed on docker containers, single thread and twelve thread, with Lambda serverless functions in both a cold state and a warm state where resources have already been provisioned for the functions.

The testing method measures the response times for different memory allocations in the warm and cold states and then plots them to see how they correlate. With a warm start the lambda system started with much faster run times at lower memory capacities, and better than the docker systems at the same memory capacity, however at higher capacities the Docker single thread performed better than the lambda.

3. STUDY DESIGN

In this section, we describe how the study is designed towards comparing the responding performance of microservice and serviceless systems. We aim at comparing microservices implemented with docker containers deployed on AWS EC2 instances, against microservices deployed as serverless functions on AWS Lambda. Both systems are versions of the same application; one of them is a dockerized microservice container, while the other is a collection of lambda functions, that is, a serverless function that only exists through execution and requires its code to be in a stateless form. Each one performs the same functionality as one of the individual services, and they are all exposed using the same API gateway instance.

The study has established certain metrics to assess the outcome, which is also aligned with the research questions of the article:

- **Response time** measured in milliseconds for each functionality in the system during tests, given a period of time.
- **Cost** for running both systems on AWS.

3.1 Experiment design

To compare microservice and serverless applications, we chose a microservice-based application that can be deployed on docker containers and transformed into a serverless-based application. The chosen application must be built with a microservice architecture that can be divided into independent serverless functions and written in a programming language compatible with the AWS Lambda environment among the following:

- Node.js 16/14/12
- Python 3.9/3.8/3.7
- Java 11/8
- Ruby 2.7
- .NET Core 3.1/.NET 6/.NET 5
- Go 1.x

In addition to this, the application must be event-driven to match the finite nature of the serverless lambda functions. The code cannot have any recursion as this leads to an increased cost due to the extra invocations of the function and the cost for the duration of each extra function. The code per each function should be relatively simple, with each function being easy to deploy and scale up, and to avoid the maximum execution time, concurrent invocations, and memory usage. The application should not rely on internal states.

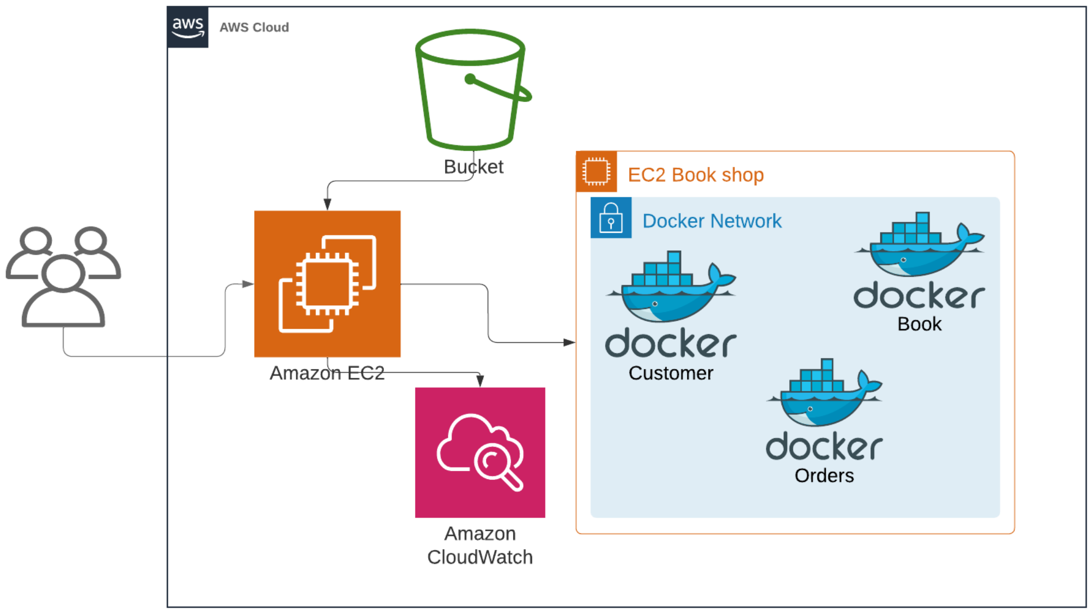
We selected the Bookshop application `https://github.com/happy-bhesdadiya/microservices` as the testbench of this study. The Bookshop application is a demonstration of microservices and consists of three separate services that track customer information, book information, and order details including information about which customer bought which book. These services are written in NodeJS using the express framework and interact with the same MongoDB database instance, which is a popular and widely used document-oriented database system that does not rely on traditional tabular relations (a connection between two tables of data) used in relational databases like SQL. The order service retrieves information from the other two services using HTTP (HyperText Transfer Protocol) method calls.

The EC2 deployed version (See Figure 3.1a) consists of three loosely-coupled microservices that operate within the same docker network and can send requests to one another, with external communication going through ports that are exposed via the EC2 instance's public IPv4 address. The instance is also connected to Amazon Cloudwatch for metrics and an S3 bucket for storage. The decision to modify the original application to run in docker containers came from difficulties in installing the required nodeJS modules in the EC2 environment.

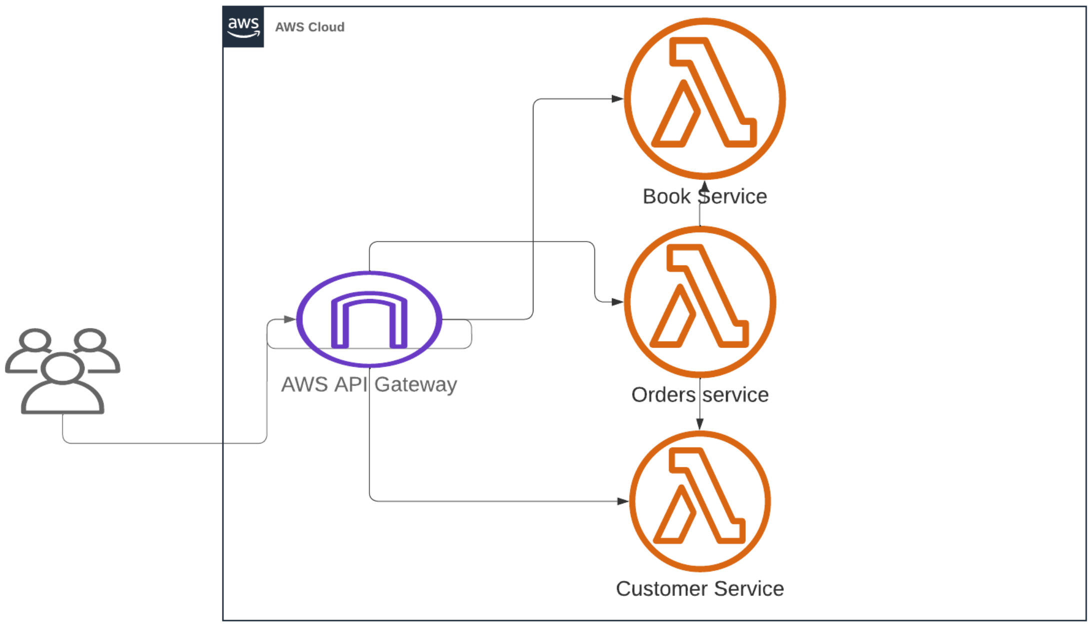
The Lambda application is made of three different Lambda functions that each represent one of the microservices from the original application (See Figure 3.1b). Each of them is exposed externally through endpoints routed through an instance of AWS API gateway. These communicate directly by using execution links. Each serverless code excerpt was deployed on a separate Lambda instance that was running on a NodeJS 12.x run-time. Serverless is not directly compatible with ExpressJS, which is the used framework in the original application, so a JS package called Serverless Express is used to wrap each function in a handler.

In order to assess the performance of both systems, unit tests were conducted on both the microservice and serverless versions using Locust.py to simulate mock user requests. The tests were performed with 10 concurrent users first, then increased to 50, 100, 500, and finally 1000, each for approximately 30 seconds. Although the tests were virtually identical, they had different endpoints due to the differences between the systems.

During the test, each simulated user was lined up at the start, and they frequently made



(a) *Concept map depicting the AWS (Amazon Web Services) EC2 (Elastic Computing Container) architecture.*



(b) *Concept map depicting the AWS (Amazon Web Services) Lambda serverless event-driven compute service.*

Figure 3.1. *Architecture comparison between AWS EC2 (3.1a) and the Lambda serverless 3.1b. The former consists of three loosely-coupled services that are orchestrated using docker containers operating on a docker network that is exposed to external traffic via the EC2 container's public IP (Internet Protocol) address by binding certain internal ports to external ports. The container is monitored using the AWS Cloudwatch tool for debugging purposes. A storage tool is used to upload the relevant files to the container. The latter service consists of three different Lambda serverless functions that are independent of each other but can communicate via direct invocation. These are exposed to external traffic using AWS' API (Application Programming Interface) gateway.*

http requests with different weights. When a new entry was created in the MongoDB persistence storage during the POST actions, random strings were generated, and when querying the ID of the returned items from the GET requests, the IDs were stored and then accessed. The requests were made in a random order. The code for the mock requests is as follows:

```
from locust import HttpUser, task, between
import random
import string
import json
from random import randint
from random import randrange
import datetime
global book_res
global customer_res
global customerid
global bookid
#orders_res
def random_date_generator():
    temp = randint(0, 4)
    random_y = 2000 + temp * 10 + randint(0, 9)
    random_m = randint(1, 12)
    random_d = randint(1, 28)
    # to have only reasonable dates
    return str(random_y) + "-" + str(random_m) + "-" + str(random_d)

def random_string_generator():
    return random.choices(string.ascii_lowercase, k=5)
class Requests (HttpUser):
    wait_time = between(0.5, 1)
    @task(1)
 #tags that allow locust to find tasks
    def create_book(self):
 #generate random data to be posted
        title = random_string_generator()
        author = random_string_generator()
        self.client.post("/book", json={"title":
           title[0], "author": author[0]})

    @task(2)
```

```
def get_books(self):
     global book_res
     x = self.client.get("/books")
     book_res = json.loads(x.text)

@task(3)
def search_books(self):


     global bookid
     bookid = book_res[0]
     self.client.get("/book/" + str(bookid["_id"]))
@task(4)
def create_customer(self):
        name = random_string_generator()
        address = random_string_generator()
        age= randint(16,35)
        self.client.post("/customer",
    json={"name": name[0], "age":age,
 "address": address[0]})
@task(5)
def get_customers(self):
    global customer_res
    x = self.client.get("/customers")
    customer_res = json.loads(x.text)
@task(6)
def search_customers(self):
    global customerid
    customer_res
    customerid = customer_res[0]
    self.client.get("/customer/"+
str(customerid["_id"]))
@task(7)
def create_order(self):
    date = random_date_generator()
    self.client.post("/order",
        json={"customerID": customerid["_id"],
        "bookID": bookid["_id"],
        "initialDate":date})
@task(8)
def get_order(self):
   global orders_res
   x = self.client.get("/orders")
   orders_res = json.loads(x.text)
```

3.2 Data Analysis

The Locust tool was used to collect execution and performance data for the two systems under test. The tool was ran on a local machine, and the URL of each system was passed to the tool. The tool created the specified number of users at a rate of x users per second until the desired number was reached. The users continually executed tasks specified in a text file, and the tool generated a table and graph in HTML format with raw data in CSV format after the tests stopped.

To analyze the results, the data generated by the tool was plotted on a graph at different intervals (5, 10, 15, 20, 25, and 30 seconds) to compare the performance of the two systems and identify trends as the number of concurrent users increased over time.

The response time and the cost of operating both systems are compared against each other. While the response time is measured in milliseconds, the cost is calculated by applying the number of executions to the individual cost per execution. The cost of the EC2 instance is defined per on-demand instance hour. The precise number of function requests made and computing hours used is taken from AWS' billing center for the month when the tests were carried out. The Amazon billing tool was used to obtain the total number of times the Lambda functions were triggered in a month, and this was then applied alongside the average duration of the request and the cost per individual request.

4. RESULTS

In this chapter, we present the results towards answering the research questions stated earlier in the paper in terms of the performance and cost comparison between AWS EC2 (microservice) and AWS Lambda (serverless) in the form of graphs and tables with the rationale explained for each, and any anomalies or unexpected results.

4.1 RQ1. What is the difference in response time?

In order to compare the performance of the two architectural styles, first we simulate 10 concurrent users using the bookstore application in both AWS EC2 and AWS Lambda (see Figure 4.1); we ran the simulation 10 times in order to avoid potential external influences. On each simulation, we record the response time at 5, 10, 15, 20, 25, and 30 seconds respectively.

We can observe from the figures above and below, regarding the medians of both parties, their performances in terms of response time are very similar, and both are under 1000 milliseconds, although the performance of the AWS EC2 is slightly better than that of Lambda in the range of 0 to 15 seconds. However, an interesting phenomenon is that, regarding the 95th Percentile, the response time of Lambda is more than six times longer than that of EC2 in the range of 5 to 10 seconds. Meaning that, at least, 5% of the users shall experience a 7-second long latency when sending a request to Lamdba. However, when the application has been started for longer ($\geq$ 25 seconds) the 95th percentile performance of Lamdba improves to nearly the level of AWS EC2.

Furthermore, we also conduct the same simulation experiments for 50, 100, 500, and 1000 concurrent users in order to verify the consistency of the previous comparison. The comparative results are shown in Figure 4.2. In terms of the performance median, we can easily observe that Lambda often has a longer response time in the beginning (5 - 15 seconds) than EC2 and then improve to a similar level of EC2 after this period. Such a phenomenon does not stand when the user number increases up to 500, where the performance of Lamdba exceeds that of EC2 by more than 10 to 30 times after the first 20 seconds "cold start".

On the other hand, regarding the 95th percentile performance with 50 and 100 concurrent

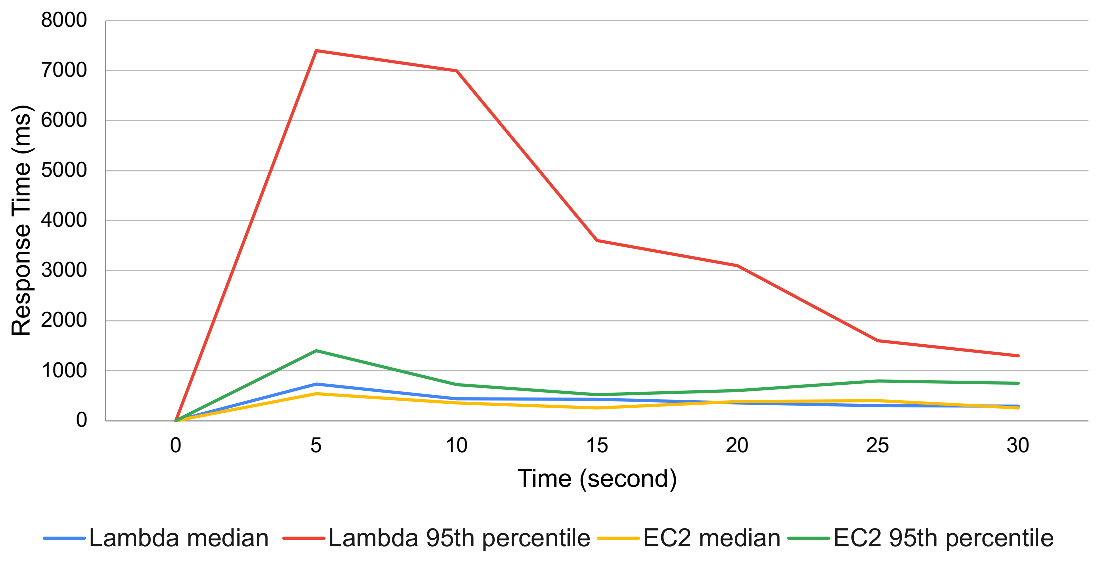


Figure 4.1. *Performance Comparision between AWS EC2 and AWS Lambda in miliseconds (10 Concurrent Users) [47]*

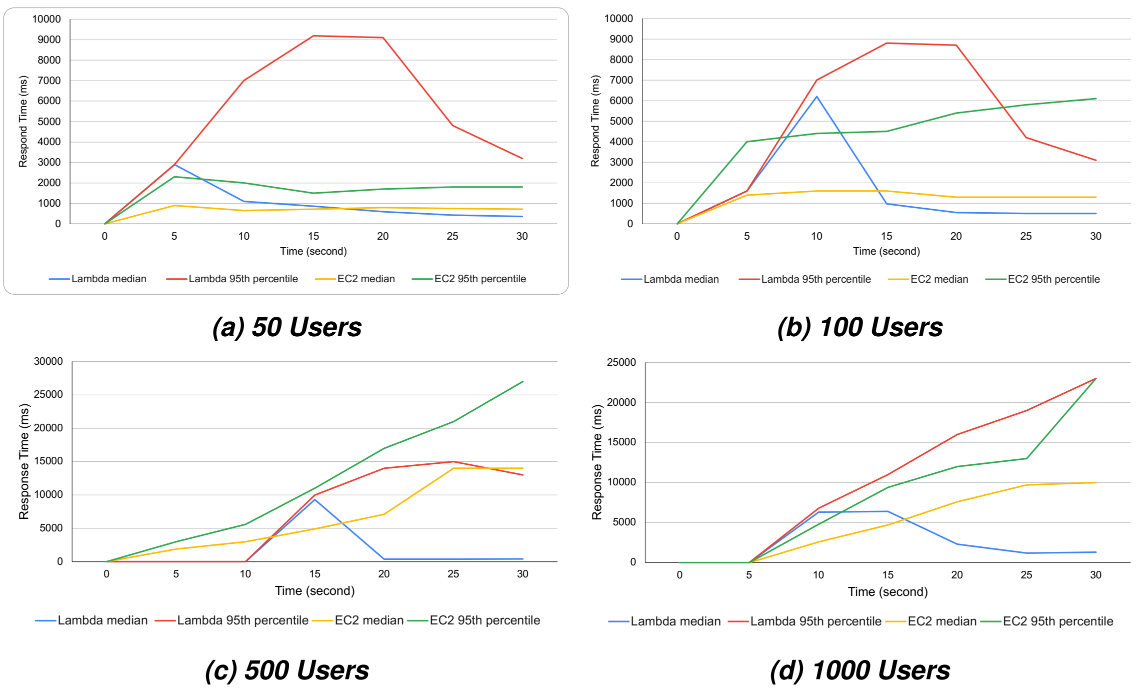


Figure 4.2. *Performance Comparison with Different User Numbers. The Y axis denotes the average response time in milliseconds, while the X axis denotes the time interval during the execution of the test that these averages happened at [47].*

users, the significant lagging phenomenon for Lambda persists for the first 20 seconds. The maximum response time can reach 9000 milliseconds. However, similar to the situation with 10 users, the 95th percentile performance of Lambda starts to improve after the 20 seconds "cold start", but still cannot reach the level of EC2. To be noted, starting from 100 concurrent users, the 95th percentile performance of EC2 decreases in performance when the user number grows. Especially, with more than 500 concurrent users, the 95th percentile response time of EC2 grows almost exponentially with the experiment time.

Other phenomena to be observed is that the peak in the lambda response times shifts to longer times in the test duration as the number of users increases and is at its highest when the number of concurrent users is at 100.

The results of these tests are in-line with the results observed in some of the works discussed earlier in earlier chapters, like the cold start latency [48], and the larger spikes in latency that come with increased concurrent users [46].

4.2 RQ2. What is the difference in cost?

Regarding the costs of the two architectural setups, we compare the potential cost when holding the same amount of requests frequency. Meanwhile, we also compare the cost difference in terms of different request rates.

Firstly, we calculate the potential monthly cost of EC2 service by taking into account the less costly example solutions. By exploring the "Savings Plans" `https://aws.amazon.com/savingsplans/compute-pricing`, we select the region of "US EAST (N. Virginia)" with the shared-tendency Linux operating system with the payment options of 1-year term length and "no upfront". For such a basic configuration, the on-demand hourly cost rate for a "t2.micro" instance (1 vCPUs, 1GB memory, Low to Moderate network performance, EBS storage) is 0.0116 USD. Therefore, the basic monthly payment for such an EC2 instance shall be $0.0116 \times 24 \times 30 = 8.352$ USD/month.

On the other hand, we calculate the cost of AWS Lambda service by taking into account the same configuration. We adopt the "US EAST (N. Virginia)" region as the default setting with which the monthly compute price is $1.67e{-}5$ USD/GB-second. With the Lambda function executed n times per month and running for 10ms each time, the monthly compute charges will be $n \times 1.67e - 5 \times 0.01$. Meanwhile, the basic monthly request price is 0.2 USD per million requests for the starting 6 billion GB-seconds month. Therefore the monthly request charges shall be calculated as $n \times 0.2 \times 1e - 6$. Therefore, the total Lambda function monthly charge is the sum of the compute charge and the request charge when the cost has a linear relation to the number of monthly requests. These calculations are summarized in Table 4.1.

As shown in Figure 4.3, we can observe the relatedness of the monthly costs of both

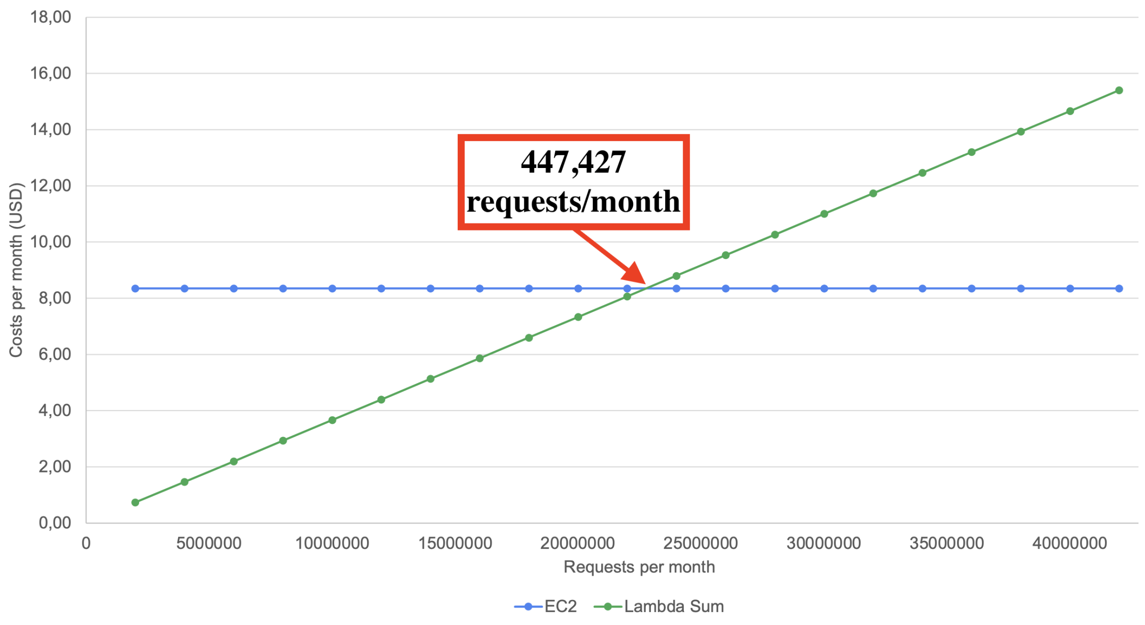


Figure 4.3. *Costs Comparison of AWS EC2 and Lambda. The intersection is the point where the expenses of running the Lambda services are as great as running the EC2 service for one month, and beyond it that it becomes more expensive. The cost of the EC2 remains static regardless of the amount of internet traffic that it receives.*

EC2 and Lambda where the intersection point can be easily calculated. For a simplified application scenario like the above-mentioned configuration, when a service receives no more than 447,427 requests per month, Lambda is a better option for savings.

Compared to other comparisons of monthly costs vs requests [43], the configurations used here have a higher threshold where the lambda solution becomes more expensive than the EC2 solution.

Table 4.1. *Cost Calculation for EC2 Vs Lambda. The measurement of the EC2 usage is the number of hours that the system is online per month, which is estimated from the cost per hour's usage by the number of hours it is used in a day and the number of days it is used per month.*

	EC2	**Lambda**
Measurement Unit	number hours per month	execution duration per month
Estimation Formula (per month)	hourly cost $\times$ #hours per day $\times$ #days per month	#executions $\times$ price per request
Cost for Basic Configurations	$0.0116 \times 24 \times 30 = 8.352$ USD/month	$n \times (2 \times 10^{-7})$;
Calculation Tool	AWS Billing center[1]	AWS Billing center/dashbird.io Lambda cost calculator[2]

The measurement for the usage of lambda functions is the duration time per each invocation in a month, multiplied by the number of times in a month that the function is invoked beyond the free tier of 1,000,000 executions. The calculation tools used in this are Amazon's own billing centers and in addition, a free calculation tool to estimate what the costs of the invocations below the 1,000,000 threshold are when the free tier is not in use. n is considered as the number of requests per month.

5. DISCUSSION

This chapter discusses the results from the previous chapter to draw conclusions from them, and reflects on the methods used to achieve them. The chapter also contains some suggestions on how the experiment could be expanded upon in future iterations and what factors should be take into consideration that could lead to different results.

In this study, we have performed a comparison between Microservices and Serverless computing by deploying two functionally identical applications using different architectural methods, in this case, for AWS EC2 and AWS Lambda. After deploying these systems we then developed a test that was designed to simulate real internet traffic by generating mock users in large quantities (10, 50, 100, 500, 1000) using the locust.py testing tool and comparing the aggregated response times for the two systems against each other. In addition to this, we analyzed the cost for both systems for the duration of the performance of the tests. The results are represented in the above graphs which compare the median and 95th percentile of the response time per request at different intervals during the 30-second test duration.

The cost of the Lambda system was calculated by multiplying the price of an individual request (2×10^{-7} USD) by the number of requests (4011 requests) and then adding the cost of the duration of the function execution time (0.00001667 USD per GB-second) multiplied by the average execution time (1996 milliseconds). By using this formula, the cost equates to 0.03 USD As for the cost of the EC2 system. This was calculated by the number of computing hours used multiplied by the cost per computing hour (0.0116 USD). Some constraints in this study relate to the environments in which the systems were deployed. In order to deploy the system to the EC2 container it was necessary to dockerize the application as there were difficulties installing the prerequisite tools and frameworks, and orchestrating these through docker.

Regarding the response time comparison outcomes (RQ1), we can observe the "cold start" issue for the AWS Lambda service in terms of both the median and 95th Percentile (see Figure 4.2d). Such a phenomenon is due to the fact that the first request for a new Lambda worker needs to find a space in the EC2 fleet to allocate and initialize. According to an analysis of production Lambda workloads provided by AWS Compute Blog, cold starts typically occur in under 1% of invocations [49]. Such an inference can be seen as supported by our outcomes showing that the "cold start" issue for the 95th percentile is more obvious and somehow insufferable due to the lagging experience. Several studies have provided potential solutions for solving the "cold start" issue of Lambda, e.g., by reducing container preparation and function loading delay, invoking function periodically preventing cold functions, using application knowledge on the function composition, etc. [50, 51].

With a growing number of concurrent users, the 95 percentile performance deteriorates significantly for both EC2 and Lambda. The reason is likely due to the selected experiment configuration with only limited resources allocated. The limitation shall be addressed in future studies by using a higher level of configuration settings and testing with a larger number of concurrent users.

On the other hand, regarding the cost comparison (RQ2), we adopted a basic application scenario for both EC2 and Lambda and found out that, when the number of requests per month is under a certain level, (in this scenario, 447,427 see Figure 4.3) Lambda is a more cost-saving option. Due to the various "saving plans" and configuration options provided by AWS, the "sweet spot" for switching services shall inevitably vary. Considering large search engine companies process about 400k searches in about 10 seconds, their preferred option shall still be EC2 with a fixed monthly quote instead of Lambda, though the eventual number will largely exceed the number for our bookstore demo scenario. To such an end, the future shall be conducted towards a more comprehensive cost calculation model with a set of critical cost parameters taken into account.

In future works, the scope of the study could be broadened in multiple ways. For example, using different Faas solutions from multiple cloud vendors in addition to different platform solutions for microservices could have a significant impact on the results with regard to both RQ1 and RQ2. Due to numerous differences between the solutions, such as:

- increased memory capacities
- different payment plans
- selection of operating systems to run a container on
- different compatible programming languages/frameworks
- different programming models (JSON objects vs triggers and bindings).

In addition, by expanding to incorporate these different solutions can give a deeper insight into the potential difficulties that companies could face when considering migration, and help to clarify if certain issues are unique to that particular vendor's solution or if it is an issue that is universal to all FaaS solutions and an unavoidable consequence of adopting that approach.

The difference in payment plans can make it so that one solution is more appropriate for a given application than others, due to the number of invocations and the frequency of those invocations vs the cost of each individual invocation per month and the duration of these invocations. For different cloud vendors the number of free invocations per month, and the cost beyond that means that the intersection for different Serverless solutions with a platform like EC2 can vary greatly.

Challenges encountered during this work occurred when finding the right application to use as the basis of the system, as some of the previous iterations turned out not to be compatible with the Lambda system, due to reasons such as in one example the application relying on an in-memory database. The next challenge was that the application decided upon needed modification to work on lambda as it relied on a package called expressJS which is a server framework. In order to get it to function in a serverless environment another framework called serverless-http was employed to wrap each serverless function in a wrapper with minimal changes to the code. The next challenge was in setting up the environment on the EC2 container. Due to difficulties in installing the necessary prerequisites in the container, the application was dockerised to make installing dependencies and networking between the services easier.

The collecting and analysis of the results was simplified greatly by the testing tools employed. The only work that needed to be done was to export the data and to plot it ourselves. As for testing the cost it took more effort to develop the model used to calculate the intersection given the different ways that the two systems are billed from the vendor. The model developed is similar to that of the one in the earlier mentioned work on the economics of serverless [43].

6. THREATS TO VALIDITY

This section addresses the threats to the validity of our research. We consider Wohlin's taxonomy [52] for this. There are four different types of threats to validity that we cover, construct validity, internal validity, external validity, and conclusion validity. Construct validity refers to the extent that our test accurately assesses what it is intended to, internal validity concerns assumptions being made about experimental conditions and the validity of those claims. External validity is about how the findings relate to real-world scenarios and conclusion validity the author bias.

Construct Validity: We have implemented two system versions, one for each approach. We used development frameworks that are used in the enterprise architecture to develop such systems, each system with comparable resources. Moreover, we used conventional practices aligning with the particular approach to developing these systems. Regarding the system size, it is limited but sufficient to demonstrate the differences. The simulation traffic has been fabricated and reflects conventional system testing approaches. To measure performance, we used the established tool, locust.py, to mitigate inaccurate methods. The format of testing and measurement could present a validity threat, however, we used established practices and tools to mitigate these construct threats.

Internal Validity: The first potential threat to internal validity is related to the fact that one author developed both system versions to ensure that they have the same functionality and comparable amount of computing resources. There may be some unintentional biases or errors in the development process that could affect the results. However, other authors verified the implementation. Another potential threat is related to the timing deviation in the measurements. To limit this, the experiment was repeated 10 times for 30 seconds each, and the values were averaged.

External Validity: Regarding the case study, we have used a small system benchmark; however, all the design principles remain the same for arbitrary sizes. The performance evaluation must be assumed in the context of a small system limiting the perspective on sample data access operations, not involving complex business logic or data routings. The motivation of the study is not to derive exact costs but to draw the relative difference between the two considered approaches, and the findings render themselves significant.

Conclusion Validity: We minimized the risk of author bias in the study by eliminating any potential mechanisms that could influence the performance of the two cloud models. Furthermore, we employed a small system with two implementations to compare the performance and costs of two cloud design approaches. As a result, our study findings demonstrate a substantial cost difference and a noticeable performance variation as the number of users increases.

7. CONCLUSION

The goal of this thesis was to answer the two questions on the performance and costs of microservices vs serverless functions. The tests and analysis carried out concluded that in the case of a simple event-driven application that serverless FaaS perform better after initial cold starts and cost less to maintain than microservices on an IaaS system. Therefore the recommendation for businesses looking to reduce costs and improve performance is to use serverless solutions. To achieve these results, we used the demo application to develop a microservices solution and a serverless solution. From there we used mock user requests at different quantities of concurrent users to test the response times of the two systems and compare the medians of the two. Then the costs from running the tests were applied to a model in order to make them compatible.

In future, the methodology could be expanded to include more configurations of the two systems at different tiers offered by the cloud vendor. In addition, more cloud platforms could be included to give insight to any peculiarities present in the vendor used that could influence the results. Changes made to the demo application were done to ease the deployment and the original functionality was kept intact as much as possible.

In conclusion, no major issues were faced with the deployment of either system or in the development of models to analyse the data. The two research questions that were set at the start of this work were answered in a satisfactory manner.

REFERENCES

[1] Taibi, D., Lenarduzzi, V. and Pahl, C. Architectural patterns for microservices: a systematic mapping study. *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. SciTePress. 2018.

[2] Taibi, D., Lenarduzzi, V. and Pahl, C. Microservices anti-patterns: A taxonomy. *Microservices: Science and Engineering* (2020), pp. 111–128.

[3] Taibi, D. and Systä, K. From monolithic systems to microservices: A decomposition framework based on process mining. (2019).

[4] Lenarduzzi, V., Lomio, F., Saarimäki, N. and Taibi, D. Does migrating a monolithic system to microservices decrease the technical debt?: *Journal of Systems and Software* 169 (2020), p. 110710.

[5] Cerny, T., Abdelfattah, A. S., Bushong, V., Al Maruf, A. and Taibi, D. Microservice architecture reconstruction and visualization techniques: A review. *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE. 2022, pp. 39–48.

[6] Bushong, V., Das, D. and Cernỳ, T. Reconstructing the Holistic Architecture of Microservice Systems using Static Analysis. *CLOSER*. 2022, pp. 149–157.

[7] Cerny, T., Abdelfattah, A. S., Bushong, V., Al Maruf, A. and Taibi, D. Microvision: Static analysis-based approach to visualizing microservices in augmented reality. *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE. 2022, pp. 49–58.

[8] Taibi, D., El Ioini, N., Pahl, C. and Niederkofler, J. R. S. Patterns for serverless functions (function-as-a-service): A multivocal literature review. (2020).

[9] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L. and Pallickara, S. Serverless computing: An investigation of factors influencing microservice performance. *2018 IEEE international conference on cloud engineering (IC2E)*. IEEE. 2018, pp. 159–169.

[10] Adzic, G. and Chatley, R. Serverless computing: economic and architectural impact. *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 2017, pp. 884–889.

[11] McGrath, G. and Brenner, P. R. Serverless computing: Design, implementation, and performance. *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2017, pp. 405–410.

[12] Nupponen, J. and Taibi, D. Serverless: What it is, what to do and what not to do. *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2020, pp. 49–50.

[13] Aslanpour, M. S., Toosi, A. N., Cicconetti, C., Javadi, B., Sbarski, P., Taibi, D., Assuncao, M., Gill, S. S., Gaire, R. and Dustdar, S. Serverless edge computing: vision and challenges. *2021 Australasian Computer Science Week Multiconference*. 2021, pp. 1–10.

[14] El Ioini, N., Hästbacka, D., Pahl, C. and Taibi, D. Platforms for serverless at the edge: a review. *Advances in Service-Oriented and Cloud Computing: International Workshops of ESOCC 2020, Heraklion, Crete, Greece, September 28–30, 2020, Revised Selected Papers 8*. Springer. 2021, pp. 29–40.

[15] Jindal, A., Podolskiy, V. and Gerndt, M. Performance modeling for cloud microservice applications. *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 2019, pp. 25–32.

[16] Mahmoudi, N. and Khazaei, H. Performance modeling of serverless computing platforms. *IEEE Transactions on Cloud Computing* 10.4 (2020), pp. 2834–2847.

[17] Al-Debagy, O. and Martinek, P. A comparative review of microservices and monolithic architectures. *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE. 2018, pp. 000149–000154.

[18] Gos, K. and Zabierowski, W. The Comparison of Microservice and Monolithic Architecture. Apr. 2020, pp. 150–153. DOI: `10.1109/MEMSTECH49584.2020.9109514`.

[19] Blinowski, G., Ojdowska, A. and Przybyłek, A. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access* 10 (2022), pp. 20357–20374.

[20] Fan, C.-F., Jindal, A. and Gerndt, M. Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application. *CLOSER*. 2020, pp. 204–215.

[21] Eismann, S., Grohmann, J., Van Eyk, E., Herbst, N. and Kounev, S. Predicting the costs of serverless workflows. *Proceedings of the ACM/SPEC international conference on performance engineering*. 2020, pp. 265–276.

[22] Rodríguez-Haro, F., Freitag, F., Navarro, L., Hernánchez-sánchez, E., Farías-Mendoza, N., Guerrero-Ibáñez, J. A. and González-Potes, A. A summary of virtualization techniques. *Procedia Technology* 3 (2012). The 2012 Iberoamerican Conference on Electronics Engineering and Computer Science, pp. 267–272. ISSN: 2212-0173. DOI: `https://doi.org/10.1016/j.protcy.2012.03.029`. URL: `https://www.sciencedirect.com/science/article/pii/S2212017312002587`.

[23] Aug. 2023. URL: `https://www.docker.com/`.

[24] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. Microservices: Yesterday, Today, and Tomorrow. *Present and Ulterior Software Engineering*. Ed. by M. Mazzara and B. Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 978-3-319-67425-4. DOI: `10.1007/`

978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12.

[25] McGrath, G. and Brenner, P. R. Serverless Computing: Design, Implementation, and Performance. *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36.

[26] Vahidinia, P., Farahani, B. and Aliee, F. S. Cold Start in Serverless Computing: Current Trends and Mitigation Strategies. *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. 2020, pp. 1–7. DOI: 10.1109/COINS49042.2020.9191377.

[27] Editor. *Comparing serverless architecture providers: AWS, Azure, Google, IBM, and other Faas Vendors*. Sept. 2019. URL: https://www.altexsoft.com/blog/cloud/comparing-serverless-architecture-providers-aws-azure-google-ibm-and-other-faas-vendors/.

[28] Heinrich, R., Van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., Schulte, S. and Wettinger, J. Performance engineering for microservices: research challenges and directions. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. 2017, pp. 223–226.

[29] De Camargo, A., Salvadori, I., Mello, R. d. S. and Siqueira, F. An architecture to automate performance tests on microservices. *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*. 2016, pp. 422–429.

[30] Auer, F., Lenarduzzi, V., Felderer, M. and Taibi, D. From monolithic systems to Microservices: An assessment framework. *Information and Software Technology* 137 (2021), p. 106600.

[31] Eismann, S., Costa, D. E., Liao, L., Bezemer, C.-P., Shang, W., Hoorn, A. van and Kounev, S. A case study on the stability of performance tests for serverless applications. *Journal of Systems and Software* 189 (2022), p. 111294.

[32] Lee, H., Satyam, K. and Fox, G. Evaluation of production serverless computing environments. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 442–450.

[33] Yu, T., Liu, Q., Du, D., Xia, Y., Zang, B., Lu, Z., Yang, P., Qin, C. and Chen, H. Characterizing serverless platforms with serverlessbench. *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 30–44.

[34] Mahajan, K., Figueiredo, D., Misra, V. and Rubenstein, D. Optimal Pricing for Serverless Computing. *2019 IEEE Global Communications Conference (GLOBECOM)*. 2019, pp. 1–6. DOI: 10.1109/GLOBECOM38437.2019.9013156.

[35] Zambrano, B. *Serverless Design Patterns and Best Practices*. 1st ed. Packt Publishing, 2018.

[36] Katzer, J. *Learning Serverless*. O'Reilly Media Inc, 2020.

[37] Chapin, J. *Programming AWS Lambda : build and deploy serverless applications with Java*. 1st edition. Sebastopol, California: O'Reilly, 2020.

[38] Patterson, S. *Learn AWS Serverless Computing: A Beginner's Guide to Using AWS Lambda, Amazon API Gateway, and Services from Amazon Web Services*. Birmingham: Packt Publishing, Limited, 2019.

[39] Simovic, A. and Stojanovic, S. *Serverless Applications with Node.js*. 1st ed. Manning Publications, 2019.

[40] Zanon, D. *Building serverless web applications : build scalable web apps using Serverless Framework on AWS*. 1st ed. Birmingham, England: Packt Publishing, 2017.

[41] Leitner, P., Wittern, E., Spillner, J. and Hummer, W. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *The Journal of systems and software* 149 (2019).

[42] Lenarduzzi, V. and Panichella, A. Serverless Testing: Tool Vendors' and Experts' Points of View. *IEEE Software* 38.1 (2021), pp. 54–60.

[43] Rodrıguez, Á., López, E. and Horrillo. *Economics of 'Serverless'*. 2018. URL: https://www.bbva.com/en/innovation/economics-of-serverless/.

[44] Canali, C., Lancellotti, R. and Pedroni, P. Microservice Performance in Container- and Function-as-a-Service Architectures. *2022 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 2022, pp. 1–6. DOI: 10.23919/SoftCOM55329.2022.9911406.

[45] Mohanty, S. K., Premsankar, G. and Francesco, M. di. An Evaluation of Open Source Serverless Computing Frameworks. *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2018, pp. 115–120. DOI: 10.1109/CloudCom2018.2018.00033.

[46] Nday, B. A., Kusuma, G. P. and Fredyan, R. Serverless utilization in microservice e-learning platform. *Procedia Computer Science* 216 (2023). 7th International Conference on Computer Science and Computational Intelligence 2022, pp. 204–212. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2022.12.128. URL: https://www.sciencedirect.com/science/article/pii/S1877050922022062.

[47] Allen Li, E. Comparing Cost and Performance of Microservices and Services in AWS:Lambda vs EC2. 2023.

[48] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L. and Pallickara, S. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018, pp. 159–169. DOI: 10.1109/IC2E.2018.00039.

[49] Accessed: 2023-07-31. URL: https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1.

[50] Vahidinia, P., Farahani, B. and Aliee, F. S. Cold start in serverless computing: Current trends and mitigation strategies. *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE. 2020, pp. 1–7.

[51] Bermbach, D., Karakaya, A.-S. and Buchholz, S. Using application knowledge to reduce cold starts in FaaS services. *Proceedings of the 35th annual ACM symposium on applied computing*. 2020, pp. 134–143.

[52] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B. and Wessln, A. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642290434.