

Architectural Implications of Serverless and Function-as-a- Service

Arkitektoniska implikationer av Serverlös Arkitektur och Function as a Service

Oscar Andell

Supervisor : John Tinnerholm
Examiner : Daniel Ståhl

External supervisor : Peter Halvarsson

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

硕士学位论文
Dissertation for Master's Degree
(工程硕士)
(Master of Engineering)

无服务器和功能服务化的架构含义
Architectural Implications of Serverless and
Function-as-a-Service
Oscar Andell 奥斯卡



哈爾濱工業大學



Linköping University

2020年6月

国内图书分类号：TP311

学校代码：10213

国际图书分类号：681

密级：公开

工程硕士学位论文
Dissertation for the Master's Degree in Engineering
(工程硕士)
(Master of Engineering)

无服务器和功能服务化的架构含义
**Architectural Implications of Serverless and
Function-as-a-Service**

硕 士 研 究 生：Oscar Andell

导 师：聂兰顺

副 导 师：Daniel Ståhl, Kristian Sandahl,
John Tinnerholm

实 习 单 位 导 师：Peter Halvarsson

申 请 学 位：工程硕士

学 科：软件工程

所 在 单 位：软件学院

答 辩 日 期：2020 年 6 月

授 予 学 位 单 位：哈尔滨工业大学

Classified Index: TP311

U.D.C: 681

Dissertation for the Master's Degree in Engineering

Architectural Implications of Serverless and Function-as-a-Service

Candidate:	Oscar Andell
Supervisor:	Lanshun Nie
Associate Supervisor:	Daniel Ståhl, Kristian Sandahl, John Tinnerholm
Industrial Supervisor:	Peter Halvarsson
Academic Degree Applied for:	Master of Engineering
Speciality:	Software Engineering
Affiliation:	School of Software
Date of Defence:	June, 2020
Degree-Conferring- Institution:	Harbin Institute of Technology

摘 要

无服务器或功能服务化（FaaS）是一种最新的架构方式，它基于抽象化基础结构管理并将其缩放到零的原则，这意味着可以动态启动和关闭应用程序实例以适应负载。这种没有闲置服务器和固有的自动缩放的概念既有好处，也有缺点。

本文对无服务器体系结构的性能和含义进行了评估，并将其与所谓的整体架构进行了对比，在 FaaS 平台 Microsoft Azure Functions 以及 PaaS 平台 Azure Web App 上实现并部署了三种不同的架构，通过测试冷启动，响应时间和被测架构的缩放比例以及观察特性（例如成本和供应商锁定）的实验得出了结果。结果表明，无服务器架构虽然受到诸如供应商锁定和冷启动之类的缺陷的影响，但它却为系统带来了一些好处，例如可靠性和降低成本。

关键词：功能服务化，无服务器，软件架构，冷启动，微服务

Abstract

Serverless or Function-as-a-Service (FaaS) is a recent architectural style that is based on the principles of abstracting infrastructure management and scaling to zero, meaning application instances are dynamically started and shut down to accommodate load. This concept of no idling servers and inherent autoscaling comes with benefits but also drawbacks.

This study presents an evaluation of the performance and implications of the serverless architecture and contrasts it with the so-called monolith architectures. Three distinct architectures are implemented and deployed on the FaaS platform Microsoft Azure Functions as well as the PaaS platform Azure Web App. Results were produced through experiments measuring cold starts, response times, and scaling for the tested architectures as well as observations of traits such as cost and vendor lock-in. The results indicate that serverless architectures, while it is subjected to drawbacks such as vendor lock-in and cold starts, provides several benefits to a system such as reliability and cost reduction.

Keywords: Function-as-a-Service; Serverless; Software Architecture; Cold Start; Microservices;

Acknowledgement

I would like to express my thanks to my examiner Daniel Ståhl and supervisor John Tinnerholm for their feedback and insights throughout the entire project. A special thanks should also go to Jesper Hölmström, Axel Löjdquist & Gustav Aaro for the support during the thesis, the last five years, as well as being my companions during our travels in Vietnam and China. Finally, I wish to thank Tong Zhang for helping me translate the thesis title and abstract to Chinese.

Glossary

API	(Application Programming Interface). An interface for communication between applications.
Artillery	Load generating tool from artillery.io
Azure	Microsoft's cloud service platform.
Azure Functions	Azure's FaaS platform. Uses <i>Function app</i> as a deployment unit. Meaning several serverless functions can scale together, share code and dependencies.
Azure Web App	Microsoft Azure PaaS platform for hosting web applications.
BaaS	(Backend-as-a-Service) Services that offer backend components such as authentication or data storage. (See Section 2.2.1)
FaaS	(Function-as-a-Service) Platform offering users to upload and deploy functions in the cloud. (See Section 2.2.1)
HTTP	(Hypertext Transfer Protocol), Request-response protocol for transferring data on the world wide web.
HTTPS	(Hypertext Transfer Protocol Secure), Encrypted version of HTTP.

Microservices	Style of software architecture where a system is composed of several loosely coupled services.
Monolith	A style of software architecture where a system consists of one potentially large executable.
PaaS	(Platform-as-a-Service) Environment for development and deployment in the cloud. It encompasses things from infrastructure such as servers and storage, to middleware and development tools. (See Section 2.2.1)
REST	(Representational State Transfer) A style of interface for communication between applications. REST services expose predefined stateless operations triggered by incoming requests.
Serverless	Can refer to FaaS, or more broadly the concept of abstracting away scaling and infrastructure management from the developer. (See Section 2.2.1)

目 录

摘 要	I
ABSTRACT	II
ACKNOWLEDGEMENT.....	III
GLOSSARY	IV
CHAPTER 1 INTRODUCTION	1
1.1 BACKGROUND	1
1.1.1 Zenon & ZenApp	2
1.2 THE PURPOSE OF THE PROJECT	2
1.3 THE STATUS OF RELATED RESEARCH.....	3
1.3.1 Related Work.....	3
1.4 DELIMITATIONS.....	6
1.5 MAIN CONTENT AND ORGANIZATION OF THE THESIS	6
CHAPTER 2 THEORY	7
2.1 MONOLITHIC & MICROSERVICE ARCHITECTURE.....	7
2.1.1 Microservices	8
2.2 SERVERLESS.....	9
2.2.1 Defining the term “Serverless”.....	10
2.2.2 Serverless Architecture	11
2.2.3 Benefits & Drawbacks.....	13
2.3 TAXONOMY OF MONOLITH, MICROSERVICE & SERVERLESS	14
2.4 FAAS PLATFORMS	15
2.5 PERFORMANCE OF SERVERLESS & WEB APPLICATIONS	16
2.5.1 Benchmarking tools	17
2.6 EMPIRICAL RESEARCH IN SOFTWARE ENGINEERING.....	18
CHAPTER 3 SYSTEM REQUIREMENT ANALYSIS.....	20
3.1 THE GOAL OF THE SYSTEM.....	20
3.2 THE FUNCTIONAL REQUIREMENTS.....	22

3.2.1 Use Case Diagram.....	22
3.3 THE NON-FUNCTIONAL REQUIREMENTS.....	23
3.4 BRIEF SUMMARY	23
CHAPTER 4 SYSTEM DESIGN.....	24
4.1 MONOLITH ARCHITECTURE	24
4.2 SERVERLESS ARCHITECTURE.....	26
4.3 BRIEF SUMMARY	27
CHAPTER 5 SYSTEM IMPLEMENTATION	28
5.1 THE ENVIRONMENT OF SYSTEM IMPLEMENTATION	28
5.1.1 Azure Functions & Serverless Implementations	30
5.1.2 Delimitations of Implementation	32
5.2 ARCHITECTURAL OVERVIEW	32
5.3 KEY PROGRAM FLOW CHARTS.....	34
CHAPTER 6 METHOD	35
6.1 HYPOTHESIS & EXPERIMENT GOAL.....	35
6.2 EXPERIMENTS.....	36
6.2.1 Use case Scenarios.....	37
6.2.2 Metrics	38
6.2.3 Experimental Design.....	39
6.2.4 Experimental Context & Systems Under Test	41
6.2.5 Instrumentation.....	42
6.2.6 Experimental Execution	43
6.3 COMPLEMENTARY OBSERVATIONS, FINDINGS & ANALYSIS.....	44
CHAPTER 7 RESULTS.....	45
7.1 EXPERIMENT 1 COLD START IMPACT	45
7.2 EXPERIMENT 2 LOAD TESTING	47
7.2.1 Scenario 1	48
7.2.2 Scenario 2.....	50
7.3 COMPLEMENTARY OBSERVATIONS & FINDINGS.....	53
7.3.1 Vendor Lock-in.....	54
7.3.2 Architecture & Extendibility	54

7.3.3 Reliability & Infrastructure Management.....	54
7.3.4 Costs & Billing	55
CHAPTER 8 DISCUSSION	56
8.1 PERFORMANCE	56
8.2 ARCHITECTURAL IMPLICATIONS OF SERVERLESS	58
8.3 PRICING & COST.....	59
8.4 COMPARISON OF MONOLITH, SERVERLESS & μ SERVERLESS	61
8.5 THREATS TO VALIDITY AND RELIABILITY	62
8.5.1 Construct Validity.....	63
8.5.2 Internal Validity.....	63
8.5.3 External Validity.....	64
8.5.4 Reliability.....	64
8.6 WORK IN A WIDER CONTEXT.....	64
CONCLUSION.....	66
FUTURE WORK.....	67
REFERENCES	68
APPENDIX A.....	72
A.1 DEPLOYMENT CONFIGURATION	72
A.2 EXPERIMENT 2, FULL RESULTS	73
A.3 DEFINITION OF IDENTIFIED VARIABLES.....	75

Chapter 1 Introduction

1.1 Background

Serverless or Function-as-a-Service (FaaS) is a new generation of cloud-based architecture that has gained popularity in the later years[1, 2]. It follows the trend of “*microservices*” where applications are built as small independent services instead of a single “*monolith*” executable. *Serverless* takes this concept even further and instead of services, applications are built by creating and connecting multiple independent cloud functions. With this new way of building software, developers write stateless, short running independent functions to be executed in the cloud, which are executed in response to triggers such as HTTP requests. These functions are automatically started, terminated, and scaled to accommodate load by the FaaS platform provider. The serverless architecture is fully dependent on cloud infrastructure and promises reduced operational cost, green computing, simpler development, and more[3]. It focuses on abstracting away all infrastructure and server management from the developer's perspective so only business logic remains.

Right now several cloud providers offer serverless functionality such as Amazon through *AWS Lambda*, Google through *Cloud Functions*, and Microsoft through *Azure functions*[1]. While this new trend in software development offers significant benefits, it does not come without its drawbacks. Considering that this novel way of software development raises many questions, especially in the area of performance and the relinquishing of control of all infrastructure to the cloud provider. This study will explore and focus on the implications of using this new type of software architecture.

In a serverless architecture, the cloud provider will provide runtime environments on-demand when functions are called. This process of allocating resources before executing a serverless function takes time and can cause performance issues in terms of increased latency. This aspect of the technology is called a “*cold start*.” In the case of user applications, research has shown that even small delay and variance in response times is noticeable to users and ultimately leads to less usage[4]. Other types of applications may be even more

sensitive to latency variance. Understanding this aspect is important when designing software systems. This thesis examines cold starts, scaling, and general performance in a serverless environment and contrasts it with the monolith approach.

1.1.1 Zenon & ZenApp

The study will be conducted in the context of developing a proof-of-concept user-service positioning application, which for this study will be referred to as ZenApp. ZenApp is proposed by Zenon, which is a consulting company in Linköping, Sweden. The application will allow users to subscribe to different services. The application will then send alerts to users if a subscribed service becomes available in their nearby area. A simple example of a service could be a carwash service. When a user is in need of a carwash, the user can subscribe to that service through the application. ZenApp will then send an alert to the user if the queue time is less than five minutes and the user is within a radius of five kilometers.

ZenApp can be seen as a generalized version of Zenon's previous Android application Blixtvakt. Blixtvakt uses a third-party weather API to alert users if a lightning strike occurs in their nearby area. The idea is to create an application where this feature can be extended to implement multiple third-party services. A more detailed description of ZenApp and the system requirements are described in Chapter 3.

While the study is anchored in this proof-of-concept system, in order to promote the generalizability of the study's findings, an abstract approach to implementing the system was chosen. Meaning many of the concepts discussed in this paper are applicable to other web applications in other contexts.

1.2 The Purpose of the Project

While there exists previous research investigating the performance of serverless and the cold-start problem[5, 6], this paper takes the approach of looking at a more complex, multilayered implementation of serverless architectures to further explore the implications and applicability in an industry context. To be able to see the performance implications of this architecture, the serverless architecture is compared

with a monolith implementation of the same application. This aim leads to the following research questions:

RQ1: What are the effects of implementing the proposed system in a serverless architecture with regards to expected response time?

SQ1: How does serverless implementation affect the latency from a user's perspective compared to a monolithic counterpart?

SQ2: What is the impact of cold versus warm starts in a serverless architecture?

SQ3: How does the serverless autoscaling during increased traffic load affect user latency?

RQ2: What are the observed implications of choosing a serverless architecture to fulfill the requirements of the system?

The thesis aim is divided into two main research questions, *RQ1* and *RQ2*. *RQ1* is further split up into three sub-questions *SQ1*, *SQ2*, and *SQ3*, each focusing on a separate area related to response times.

1.3 The Status of Related Research

Serverless and serverless architecture is an emerging topic in research[7]. There have been large investments in serverless technologies and FaaS platforms from the software industry, but extensive research in the area is missing and currently many open research problems and challenges still exist[2, 8]. This section along with *Chapter 2 Theory* covers the related research and body of knowledge laying a foundation of this thesis. This section covers the most relevant research papers related to the aim of the study and what contribution this thesis brings to the research topic.

1.3.1 Related Work

M. Villamizar et al.[9] in the paper “*Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures*” conducts a study where they evaluate the cost and performance of three distinct software architectures. These are the monolith, microservice, and serverless architectures. To be able to evaluate the implications of using respective architecture, a case study was designed where the same application

was developed in different architectures. In the study, they described the process of implementing a system in the monolith, microservice, and serverless architectures and the challenges faced. All versions of the application were deployed on Amazon Web Services. (The serverless implementation was operated by *AWS Lambda*). By running performance tests and making cost comparisons, the study concluded that using FaaS platforms such as AWS Lambda can reduce infrastructure costs by up to 77.08%. Additionally, in the case of small applications, the study found that a monolith approach is more practical since the development and deployment process of microservices and serverless architectures tend to be more complex.

Similarly, Albuquerque Jr et al.[10] perform a comparative study on Platform-as-a-Service (PaaS) and the serverless (FaaS) model. The authors developed a simple application in the microservice architecture. One version of the applications was deployed on AWS's PaaS platform and the other version on the FaaS platform *AWS lambda*. The performance between the two implementations was measured by sending a high amount of HTTP traffic to the application, triggering different functionalities of the application. With the experiments, the authors perform a performance and scalability analysis where they found that while the performance is similar between the two solutions, cold starts can have a negative impact on FaaS functions. The study also compared the cost between the two platforms and found that PaaS is more economically suitable for applications with longer or varied execution times while FaaS has a better cost-benefit of requests with short and predictable execution times.

J. Manner et al. in the paper *Cold Start Influencing Factors in Function as a Service*[5] investigated cold starts in FaaS functions. The authors presented a hypothesis of the factors that influence the severity of the cold start delay, which includes factors such as programming language, number of dependencies, package size and more. The study also investigates how to benchmark cold starts in serverless functions to get repeatable experiments and results. The authors chose to conduct the study on the platforms *AWS Lambda* and *Azure Functions* with functions calculating a recursive Fibonacci sequence. The programming languages used were JavaScript and Java, one interpreted language and one compiled. To measure the difference between cold and warm starts, the researchers performed the experiment sequentially. First triggering a cold start

followed by a warm start, then waiting for the container to shut down, and repeat the sequence. The study confirmed their hypothesis that cold starts are impacted by programming language and claimed that cold-start overhead can range from 370ms to 24 seconds depending on language and platform.

In a similar fashion, D. Jackson et al.[6] evaluated the performance of different programming languages in serverless applications. They also examined the costs of serverless functions. To test this, the researchers constructed what they call the “*Serverless Performance Framework*” which is an open-source tool that uses scheduled events to trigger the serverless functions under test, as well as calculates an estimated cost of that execution. This approach removes external latencies such as API gateways from the results. This study, like J. Manner et al. found that language runtime and platform have a significant impact on performance. They also find that the choice of language also affects cost.

An example of a complex application built with a serverless architecture comes from M. Yan et al.[11]. In the paper, the authors describe the architecture and implementation of a chatbot on the OpenWhisk platform. The chatbot used several layers of serverless functions, the first layer to convert voice to text, second to parse the text and routed the request to the appropriate serverless function in the third layer. The third layer uses several third party API:s, for example, a weather service, allowing a potential user to ask the chatbot about the weather in a particular city. The authors argued that this architecture is inherently extensible and scalable. The authors state that the performance of the chatbot prototype was not tested, however, that the expected latency would be in the order of 1-2 seconds.

What this thesis seeks to accomplish, in comparison to the mentioned research, is to go beyond the performance research with trivial applications and functions and instead evaluate a more complex implementation. While M. Yan et al. showed that complex applications can be built with serverless architecture, the performance implications have not been evaluated. By combining the aspects of the architectural research of M. Yan et al., the performance comparisons of Villamizar et al. and the study of function cold starts by J. Manner et. al., the contribution of this study is the evaluation a non-trivial proof-of-concept system built with the monolith and serverless architectural patterns, both in terms of performance and architectural implications. The analysis of the collected data

was inspired by C. Seaman et al. [12]. The authors used a mix of qualitative and quantitative methods to study communication during code inspections in a software project. In the study, the authors explore and analyze the relationship of different variables to generate hypotheses of how different variables affect the inspection process. This method of analysis was applied to the findings of this thesis to explore the implications of studied architectures.

1.4 Delimitations

The application used for evaluating the serverless architecture is a REST-API that carries out read and write operations to a database. No heavy operations or compute-intensive logic was evaluated. The system was developed in Node.js and deployed on the Microsoft's serverless platform, making the study limited to JavaScript functions deployed on Azure Functions. The justification for focusing on these technologies is discussed in Section 5.1, *The environment of system implementation*.

1.5 Main Content and Organization of the Thesis

The first chapter presents a brief background of the topic of serverless as well as the aim and research questions this study will cover. *Chapter 1* also presents related work and how it relates to the research of this thesis. *Chapter 2* presents a theoretical frame of reference for the study, covering terminology, definitions, and previous research. The evaluated implementations are detailed in *Chapters 3, 4, 5*. *Chapter 3* cover the requirements of the system, *Chapter 4*, the design and architecture, and *Chapter 5*, the technical implementation. The research method is presented and discussed in *Chapter 6*. This chapter covers the experimental designs and context. *Chapter 7* presents the results of the experiments and the study's findings on cold starts and load testing, as well as general observations and collected data. *Chapter 8* discusses the characteristics and implications of Serverless architectures. The chapter evaluates the study's findings and relates them to the proof-of-concept system, as well as applying and viewing them in a wider context. This chapter also includes a discussion about the study's validity. The final chapter presents the conclusions of the study as well as suggestions for future work.

Chapter 2 Theory

To be able to describe the serverless architecture, it needs to be contrasted to more traditional approaches to software architecture. This chapter covers the terminology and definitions of the technology that concerns this thesis. It will also serve as an informal literature review of previous research on the topic of serverless.

2.1 Monolithic & Microservice Architecture

The term “*Monolithic Architecture*” in this context refers to the definition by Martin Fowler[13], where he describes it as the traditional approach to software architecture.

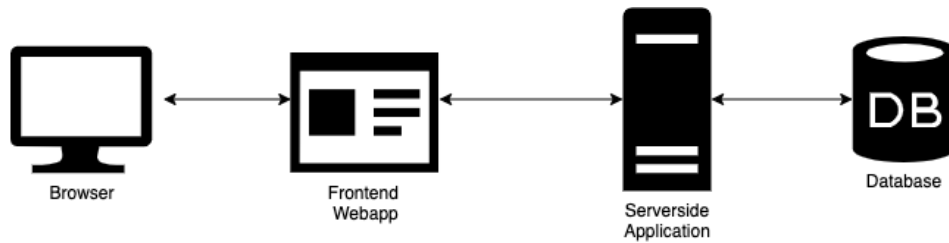


Figure 1 Monolithic Architecture

Figure 1 shows the architecture of a monolithic web application. It consists of a user interface displayed in the browser, a database to store persistent data, and a server-side application that handles requests from the frontend application and fetches data from the database. The server-side application is one, potentially large executable with a single codebase that handles all server-side logic. This according to Fowler’s definition is a “*monolith*.” The monolithic way of building an application has many benefits[14]. Developer tools such as IDEs can be focused and configured to create a single application, its simple to deploy and easy to scale.

However, the larger the application becomes, the drawbacks of the monolithic architecture become more apparent[13, 14]. Assume that the monolith server-side web application contains and offers a set of services $S = \{S_1, S_2, S_3, \dots\}$, for example, in a web store, a service S_n might be an authentication service, a search service, etc. Over time, new services are added, new developers

are assigned to the project, and thus complexity increases. Changes and bugfixes become difficult and time-consuming, slowing down development time. A large codebase can also slow down IDEs. Furthermore, building and testing the system may take significant time, further slowing down development.

Scaling is another factor that might become an issue with monolith architectures. With large amounts of traffic to the application, it might need to scale up to more instances to meet the demand. If the traffic to the services S are unevenly distributed and only a few services are used, the entire application still needs to be scaled, not only the services that are in demand, which is inefficient[9].

2.1.1 Microservices

As a response to the previously discussed inherent drawbacks of the monolith comes “*Microservices*”[13, 14]. Microservices architecture is a style of software architecture that structures an application as a bundle of loosely coupled, independently deployable services called microservices. A microservice can be described as a small application with a single responsibility, which can be scaled, tested and deployed independently of the larger system[15].

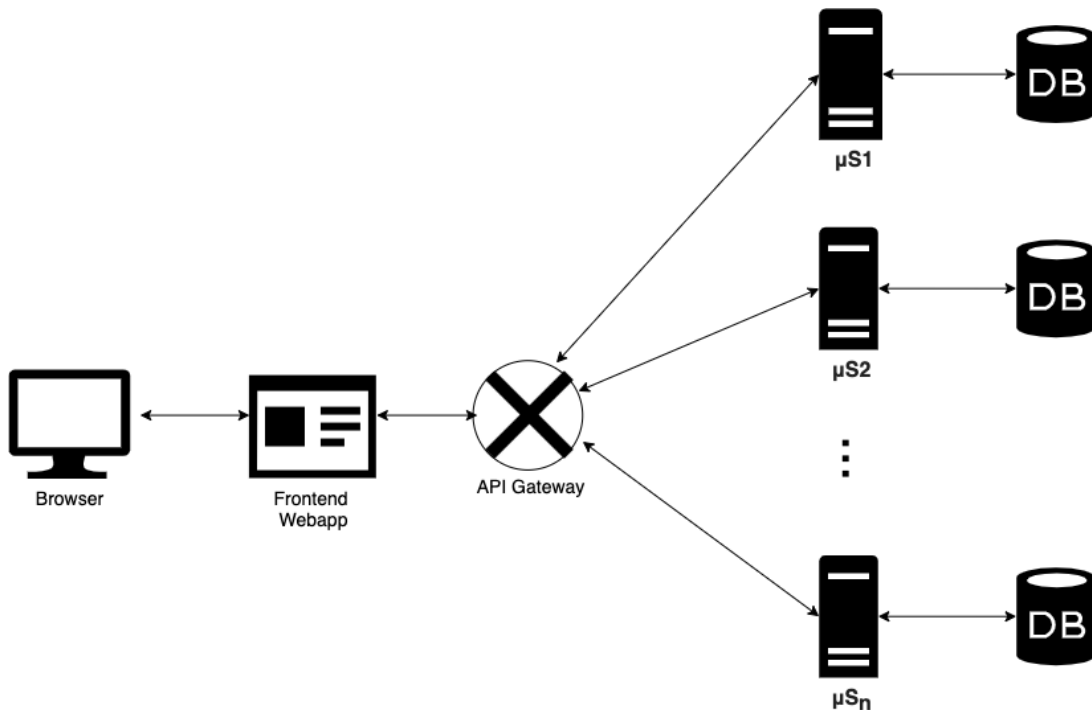


Figure 2 Microservices Example

Figure 2 illustrates a microservice architecture. In the example of the web store, the monolith server-side application is split into a set of microservices $S = \{\mu S_1, \mu S_2, \mu S_3, \dots\}$. Each microservice μS_n has its own small responsibility, offering a subset of the services $S = \{S_1, S_2, S_3, \dots\}$ [9]. In this example, the requests from the browser are instead of being sent to a single server-side application, are routed to the appropriate microservice through an API gateway. This gateway serves as an entry point to the microservice application [14].

This architecture enables increased flexibility since microservices can be built with independent teams, using the technology stack and programming language most suitable to that service [9]. Another benefit of the microservice architecture allows services to be independently scalable. This means that if one part of the system is under heavy load, it is possible to only scale the affected microservices and not the entire system, potentially reducing infrastructure cost [16].

This method of developing loosely coupled services instead of a monolith can offer more practical ways for companies to develop and manage applications with large code bases [16] and is used by companies such as LinkedIn [17] and SoundCloud [18].

However, while the microservices approach can solve many issues of the monolith architecture, it is not a fix-all solution. Microservice architecture comes at the cost of the increased effort of operating, managing deployment and scaling for multiple services in a cloud environment [9]. Instead of managing the infrastructure of one monolith, each microservice needs its own infrastructure, environment, and configuration. One potential solution to these drawbacks is *serverless architecture*.

2.2 Serverless

The novel serverless approach to microservices tries to mitigate the issues of increased infrastructure and server management by handing over all server management to a cloud provider. This section gives an overview of the term “*serverless*,” serverless architectures, and the main benefits and drawbacks of the technology.

2.2.1 Defining the term “Serverless”

Despite the name, serverless functions still run on servers, however, all server and infrastructure management are managed by a third-party. The term serverless, in the context of this thesis, will refer to what is also called *Function-as-a-Service* (FaaS) in which functions are the deployment unit i.e., what is deployed on the cloud are individual functions instead of complete applications. Several cloud providers are currently offering FaaS on their cloud platforms, among these are Amazon through *AWS Lambda*, Google through *Cloud Functions*, and Microsoft through *Azure functions*[1].

In the categorization of cloud services, FaaS would fit in the gap between Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) in terms of development control[2, 7].

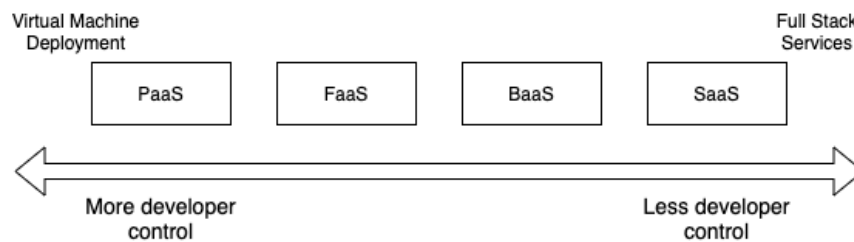


Figure 3 Overview of cloud services, adapted from [7]

PaaS allows the provisioning of servers and deployment of applications on virtual machines in the cloud. In PaaS, the developer generally has more control over infrastructure and the code that is deployed. SaaS provides users with the use of complete software and the service provider has full control of the infrastructure and source code, e.g. Gmail. FaaS is located between these (see Figure 3). In FaaS the developer does not have any control over the infrastructure, which is shared between the platform users but has control over the code deployed, which is in the form of independent stateless functions[1, 2].

Another important difference between FaaS and PaaS is scaling and cost. In PaaS, idle time is often charged but in FaaS, the functions can be scaled down to zero and be spun up at the time of use [2, 19]. Instances of FaaS functions are automatically created when the function is activated by a trigger, such as a database change or an HTTP request. FaaS functions are not designed to be long-running and have short timeouts (For the cloud provider Azure, the maximum timeout is 10 minutes). After a function has finished executing, the instance is

shut down, freeing server resources[11]. In order for the functions to be able to scale, serverless functions are essentially stateless. Variables stored in memory cannot be guaranteed to persist throughout multiple invocations of the function and thus requiring the function to be stateless or store state outside of the FaaS function instance[3].

Another cloud service closely related to serverless is Backend-as-a-Service (BaaS) [7, 20]. BaaS allows provisioning of services such as data storage or authentication from a third party, such as Google's Firebase. FaaS is a hosting environment while BaaS enables the outsourcing of application components, they both, however, can fall under the term serverless since neither requires any server management[20].

A summation of the properties of serverless functions comes from the book *What is Serverless?*[20] where the author's M. Roberts and J. Chapin states five key traits of serverless:

- No required management of infrastructure and servers. Deployment is done by uploading the function source code to the provider, the rest is handled by the provider.
- Horizontal scaling is managed by the provider and is done automatically.
- The cost is based on usage.
- Configuration of host size and instance count abstracted away from the user.
- High availability should be expected, i.e. if an underlying component fails, the provider is expected to reroute requests to another instance of the serverless function.

In summary, a serverless function is a cloud-hosted, independently scalable, stateless function that is activated and executed in response to an external trigger. For the purpose of this thesis, this is what is referred to as serverless or FaaS.

2.2.2 Serverless Architecture

Like microservices, a system built with a serverless architecture is broken down into small components, but instead of “services,” a system built with a serverless architecture will consist of many small independent, autonomous functions[2, 11].

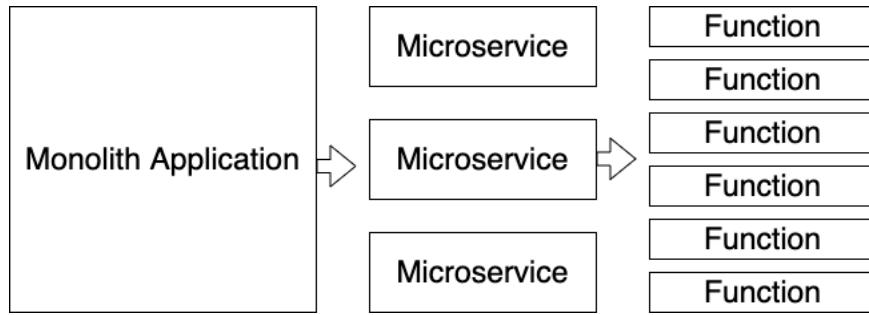


Figure 4 Architecture granularity, adapted from [21]

Figure 4 shows a visualization of the granularity of the monolith, microservice, and serverless architecture. As described in Section 2.1.1, the microservice architecture is a decomposition of a system into separate services. Serverless architectures further decomposes a system into separate serverless functions. Unlike a microservice, which can be any type of application, a serverless function contains only the code for that specific function, i.e. the boilerplate code used in for example setting up a REST API, is peeled off and handled by the FaaS provider.

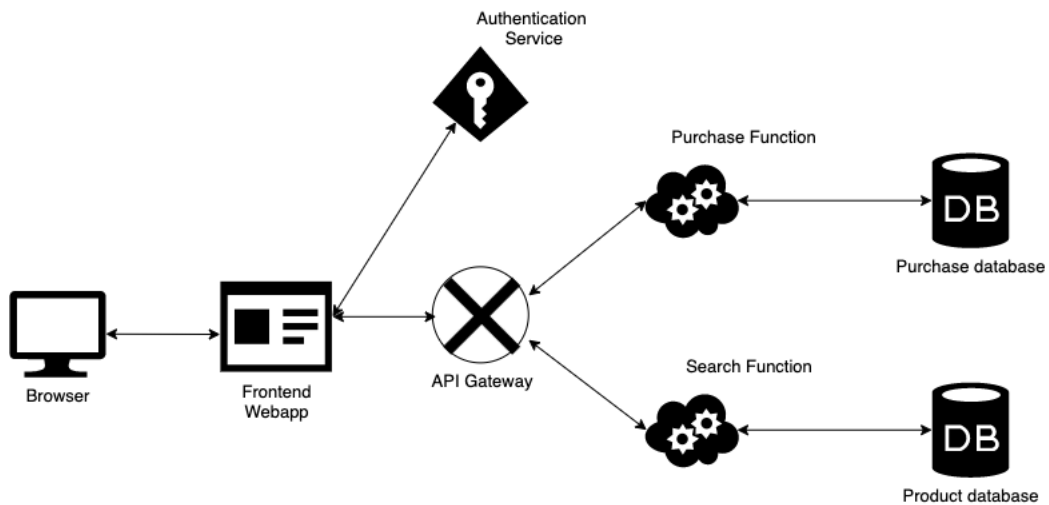


Figure 5 Example of serverless architecture, adapted from [3]

Figure 5 shows an example of a web store built with a serverless architecture. It has two FaaS functions, one containing code that handles the logic of searching for products, and the other contains code for handling purchases. The functions are placed behind an API gateway to route requests from the frontend application to the appropriate function. The functions are configured to

trigger on an HTTP request and when invoked, the cloud provider starts up an instance, runs the code, and is then shut down. This serverless architecture also uses BaaS services for authentication and database storage. Compared to the monolith and microservice approach, this architecture abstracts away everything but the business logic and allows all management of servers and scaling to be handled by third party services. More complex applications built with this architecture may utilize many serverless functions chained together in order to create complex logic and systems[11].

2.2.3 Benefits & Drawbacks

This approach to software development makes it possible to build complex applications from simple serverless functions and comes with many benefits. Mike Robert states that *"Fundamentally, FaaS is about running backend code without managing your own server systems..."*[3]. This has the added benefit of allowing developers to spend more time writing application logic and not worrying about server infrastructure and deployment since this is handled by the cloud provider[1, 2]. Infrastructure costs can also be reduced due to scaling being completely automatic, and you only pay for what you need. This has the potential to save costs, especially in the examples of occasional or inconsistent traffic, where the new instances can quickly be started to meet the traffic demand and then spun down, instead of standing idle[3, 22]. M. Villamizar et al.[9] claims that using a serverless architecture can reduce infrastructure costs by up to 77.08%.

From a wider perspective, serverless cloud computing can have a positive impact on the environment because of green computing and reduced energy consumption[3]. In a serverless context, cloud providers only allocate the amount of computation power that is needed at any specific time. This means more applications and services can share the same infrastructure and can be started and scaled when needed instead of standing idle. This reduces the need for data centers and lowers overall energy consumption, which in turn can have a positive environmental impact.

While serverless architectures have significant positive benefits, it also comes with significant drawbacks. The implementation of FaaS on different cloud service providers might be radically different and very coupled to the cloud

provider. This could make switching platforms expensive and cumbersome making vendor lock-in a drawback of the serverless approach[2, 3, 7, 20, 22]. By handing over part of the software stack you also lose full control over your application. There will be limitations in configurable parameters, and similarly, you won't be able to optimize your application for specific hardware since the underlying components are abstracted away[20]. Loss of control also affects issue resolution, any issue in the underlying infrastructure is in the hands of the service provider, meaning you have to wait for the service provider to take action[20]. Security is also a factor that you lose some control over since it is tied to the service provider[20].

An inherent drawback is the stateless nature of serverless functions, this makes dealing with application state difficult. In instances where stateful is needed the program state needs to be stored externally[20], e.g. fetching a session token from a database.

Another drawback of serverless is the concept of cold starts. A “*cold start*” in the context of FaaS refers to the process of executing a serverless function when it has scaled to zero[2], i.e. when the cloud provider starts a container to run the code. On the contrary, a “*warm start*” refers to when a serverless function is invoked while a container hosting the code is already running. The cloud provider *Microsoft Azure*[23] describes the process in steps. Before a function can be executed, a server needs to be allocated. Secondly, the runtime of the function needs to be configured and started on that server. In a warm start, the resources are already allocated, and the function can be executed significantly faster. To speed up the cold start process, Azure keeps pools of preconfigured servers with runtimes that are already running, however, loading in files and settings into the memory still causes higher latency compared to warm starts.

2.3 Taxonomy of Monolith, Microservice & Serverless

A common theme in literature is the combination of serverless and microservices. While related, they are in some orthogonal to each other. Serverless can be viewed, at least in part as a hosting and billing model, while microservices a way of structuring a system.

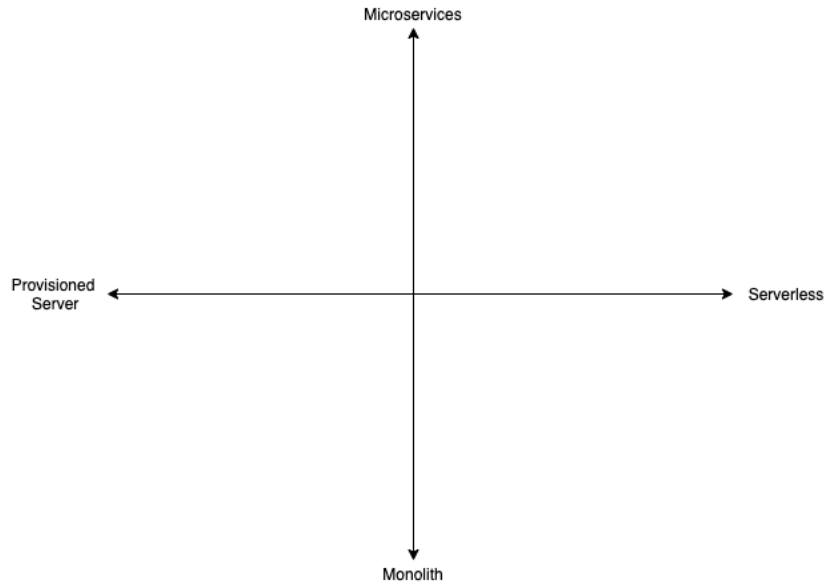


Figure 6 Microservices-Serverless 2D Space

Figure 6 shows a two-dimensional space, each axis, microservices to monolith and provisioned servers to serverless, is associated with a set of behaviors and properties. One way to view the architectures discussed in this study is that they can be placed on this plane. It is possible to design a system consisting solely of small independent serverless functions, which would place it in the top right quadrant. It is also technically possible to build a large monolith application, deployed on a serverless FaaS platform, placing it in the lower right quadrants as a serverless monolith. For this thesis, this two-dimensional plane serves as a useful tool to categorize and map properties to certain architectures and what system behaviors can be expected when situated somewhere on the plane.

2.4 FaaS Platforms

The first commercial FaaS platform was *AWS Lambda*, launched by Amazon in 2015. AWS lambda being the oldest most established FaaS platform, is the platform most prominent in academic papers[22]. Microsoft's counterpart to AWS Lambda is called *Azure Functions* and was released in 2016. More recently, Google launched the release version of its serverless computing platform in 2018, called *Google Cloud Functions*. The platforms offer similar functionality, but there are some differences in for example, allowed programming languages, cost,

monitoring and debugging[24]. The platforms are also heavily integrated with the general cloud platform of the company, meaning it is easy to hook up BaaS services such as API-gateways and databases offered by the respective platforms.

Besides the commercial platforms, there are also open-source platforms. These platforms enable running serverless functions on your own infrastructure. A few of these are Apache *OpenWhisk*, OpenFaaS, and Kubeless.

As described in Section 2.2.3, vendor lock-in is a big drawback of serverless architectures since the implementation differs between the platforms. A proposed solution to this problem is the *Serverless Framework*[25]. The Serverless Framework is a popular open-source framework for developing and deploying serverless applications on any FaaS provider. The framework offers a CLI interface for creating and configuring serverless projects, including FaaS functions and cloud infrastructure resources.

2.5 Performance of Serverless & Web Applications

The underlying infrastructure and implementation details of commercial FaaS platforms are often hidden from the user. This makes FaaS platforms like a black box and highlights the importance of performance benchmarks on these platforms. There has been recent research into the area of performance and benchmarking of FaaS platforms and FaaS functions [5, 26-28], but due to the novel nature of FaaS and serverless, platforms are evolving and updated frequently, threatening the validity of some of the research in this topic.

Research has found that performance between different platforms can vary significantly. Other aspects, such as the choice of programming language can also have a large impact on the performance and latency of a serverless function.

Cold starts and the mitigations of its effects are an ongoing research topic[5]. Research has found that cold starts can have a significant impact on latency and that the severity of the latency is also dependent on the cloud provider and the programming language used.

Another research topic is the elasticity of serverless platforms. Elasticity being “*the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner*”[29].

To be able to quantify the elasticity of serverless platforms, Kuhlenkamp et al. [26] presents an experiment design that evaluates platform with metrics such as reliability, request-response latency, and request throughput.

User perceived latency is an important part of the performance and can have an impact on the usage of a web application. I. Arpakis et al.[4] claim that in web search, as latency increases, users are less likely to perform clicks on the results. The authors claim that under 500ms, latency is not noticeable, but if the delay is over 1000ms it is very likely for users to notice the added delay.

The paper “*Defining Standards for Web Page Performance in Business Applications*”[30] by Rempel et al. define a set of standards and metrics to evaluate the performance of web applications. The authors claim that by adhering to these standards, an application would achieve high user satisfaction in terms of performance. For most basic operations, they claim that the 95th percentile target maximum latency should be less than 2 seconds. Meaning that 95% of all users should be expected to experience latency of < 2 seconds.

2.5.1 Benchmarking tools

Artillery.io[31] is a tool used and suggested in benchmarking FaaS platforms and microservices[26, 32]. Artillery is an open-source load testing and functional testing toolkit. It can simulate users to a web application by sending high amounts of network requests to a specified website or application. The CLI(Command-Line-Interface) tool allows for defining complex test scenarios where users can specify HTTP requests and payloads of data to be delivered to the application. This makes it an ideal tool to test the performance and behavior of applications that interact through a REST API. The tool is also easily scriptable and offer easy installation through the popular package manager *npm*.

Another open-source tool used for benchmarking is JMeter[33]. JMeter has also been used in web application benchmarking[9] and is a Java application for performance testing on static and dynamic web applications. Like Artillery, it can simulate high user traffic to an application and supports a wide range of network protocols, for example, HTTP, HTTPS, REST, and more.

2.6 Empirical Research in Software Engineering

One aim of this study is to adhere to the principles of empirical research in software engineering. Therefore, the informal literature review included research guidelines proposed by software engineering researchers.

B. Kitchenham et. al.[34] presents guidelines to promote the quality of empirical research in software engineering. The research guidelines cover the context and design of experiments, data collection, and presentation and interpretation of results. Experimental context is essential for reproducibility and further analysis of a research study, where details of context and circumstance need to be thoroughly described. Related research should also be defined and presented to build a collection of knowledge around the research area. The guidelines for the experimental context also describe how to ensure that the objectives of the study are properly defined, for example, if evaluating an industry technique, one needs to make sure that the version that is being evaluated is not oversimplified. The guidelines for conducting experiments highlight the importance of defining and documenting the data collection process which is an important aspect of replicability. The presentation of results is a very important part of a study, procedures of analysis and data collection need to be transparent and detailed enough so that another researcher can replicate the study or with access to the original data, draw the same conclusions as presented in the study. Finally, the authors state that the conclusions of a study should follow the results and it is of importance not to misrepresent the conclusions. Therefore, the author of a study needs to define the type of the study and to specify and be clear with the limitations and discuss the external and internal validity.

In another article, B. Kitchenham[35] argues that the role of formal experiments in the field of software engineering is overemphasized. Laboratory experiments do not give a fair representation of the actual software industry because of how experiments abstract away the industrial context and focus on isolated processes. Instead, she suggests that empirical studies in the software engineering field should instead emphasize case studies and quasi-experiments (experiments where it is not possible to assign subjects participants at random). However, Kitchenham also states that formal experiments still have value and a place in software engineering research. Proof-of-concept studies and studies where performance is measured are two of these.

P. Runeson and M. Höst[36], in the paper “*Guidelines for conducting and reporting case study research in software engineering*,” claims that case studies are a suitable research method in software engineering. This because it allows studying a case or phenomena in its natural context and seeing how it interacts in a real setting. In a case study, there are no controlled factors or controlled experiments. Instead, researchers, through a step by step process, plan, design and collect data through for example, interviews, observations, and archived data. The data is then analyzed and through a chain of evidence and triangulation, the researcher can come to a conclusion. While experiments give clear results, the authors claim experiments in software engineering are affected by many factors that might impact the replicability. Case studies, on the other hand, can produce softer results, but they can give a deeper understanding of the studied phenomena.

Chapter 3 System Requirement Analysis

The following three chapters cover the requirements, architecture, and implementation of the proof-of-concept system developed for the purpose of this thesis. This chapter covers the general functionality of the system, while Chapter 4 and Chapter 5 focuses on the development of the separate monolith and serverless architectures for the system.

3.1 The Goal of the System

The planned system can be described as *subscription services based on position*. It will allow users to subscribe to services of interest in and notify them when a particular service is available. Services are attached to a location and could, for example, be a carwash or a hair salon. An example of a use case is a user who wants to wash his or her car, the user can then subscribe to be notified when the car wash waiting time is less than five minutes.

Definitions:

- A *service* in the system refers to a service offered by the system, e.g. a hair salon, a carwash or another third-party.
- A *subscription* refers to when a user has subscribed to a service. If the service is available and the user is nearby, the user will be notified, e.g. a hair salon nearby has an available time at this moment.
- *Service criteria* – The criteria that must be fulfilled for a service to notify the subscribed user.
- *Distance of interest* – The maximum distance between a service and a user in which a user can receive a notification.

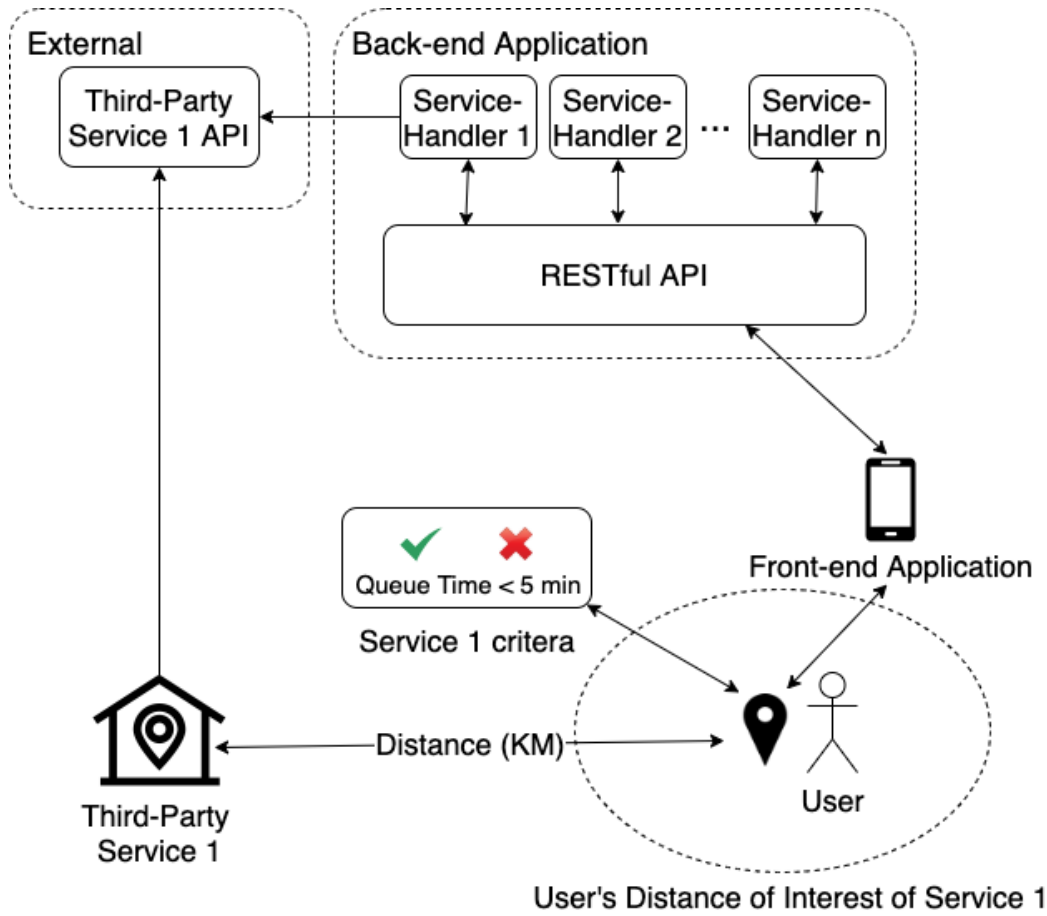


Figure 7 High-level System Overview. Adapted from [37]

Figure 1 shows a high-level overview of the proposed system. In the example, a user is subscribed to *Service 1* with a configured distance of interest. The frontend application communicates the user's position to the backend application with regular intervals. If the distance between Service 1 and the user is less than the distance of interest and that the service criteria are fulfilled (e.g. service is available or queue is less than 5 minutes) the user will receive a notification.

The backend application, which communicates with all users through a REST-API, and handles user management and subscriptions. The backend application implements a variety of services through third-party APIs and serves as an intermediary between users and external services.

The system developed during this study is a proof-of-concept implementation of the described system and although not a full-fledged feature-complete system, it still implements the requirements below. This enables the evaluation and exploration of an appropriate architecture for the future application.

3.2 The Functional Requirements

- Users should be able to subscribe and unsubscribe from different services.
- When a service becomes available, subscribed users in the area should be notified.
- Services should be able to be added and removed as available for the user.
- The system should contain functionality for adding new users.
- A service is a generic component with the following properties:
 - API for receiving incoming position, and user-configured settings.
 - API for fetching information about the service.

The requirements of the system were adapted from [37].

3.2.1 Use Case Diagram

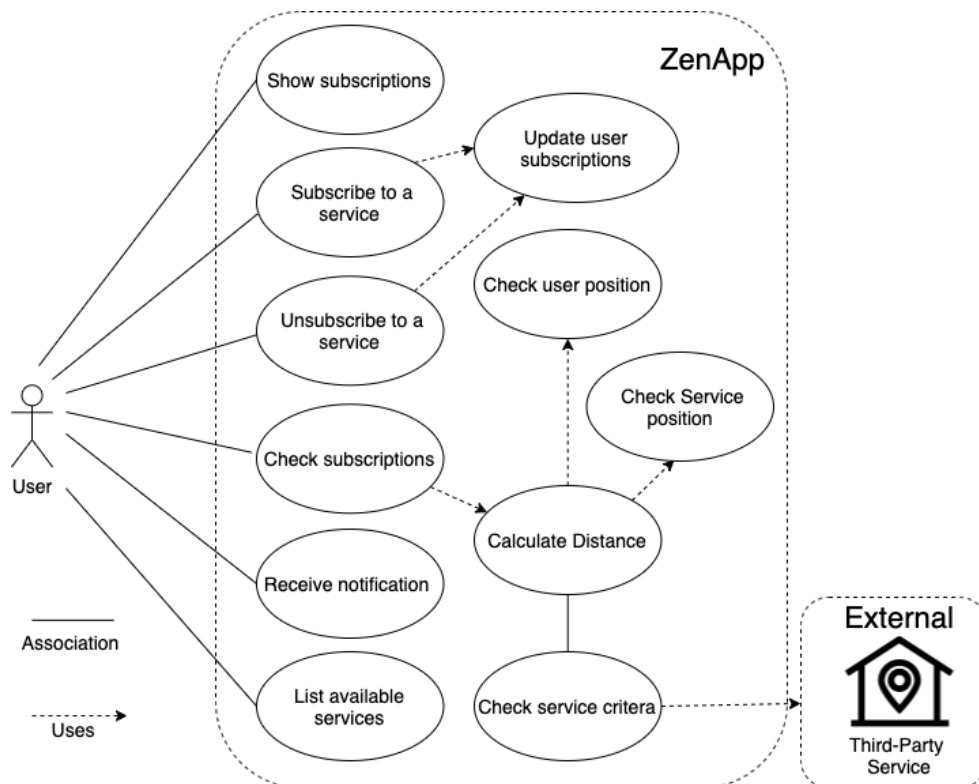


Figure 8 ZenApp Use case Diagram[37]

3.3 The Non-Functional Requirements

- Extendable - new services should be able to be added with minimal effort.
- The system should be hosted and deployed in the cloud.
- The system should be implemented in JavaScript and the Node.js runtime.
- The system should enable response time measurements.
- The system should use a REST-API for communication with clients.

3.4 Brief Summary

This chapter has given an overview of the proof-of-concept web application developed for this thesis. The system was developed with a set of functional and non-functional requirements to create comparable monolith and serverless implementations of the same system. An overview of the system features and uses is showcased in Figure 8. The design and implementations of the different architectures are detailed in the following chapters.

Chapter 4 System Design

This chapter covers the high-level design and architecture of the implementations covered by this thesis. The monolith system was designed in cooperation with J. Holmström, who evaluates the implications of distributed data in the microservice architecture[38]. From the implementation of the monolith, a serverless design of the same system created.

4.1 Monolith Architecture

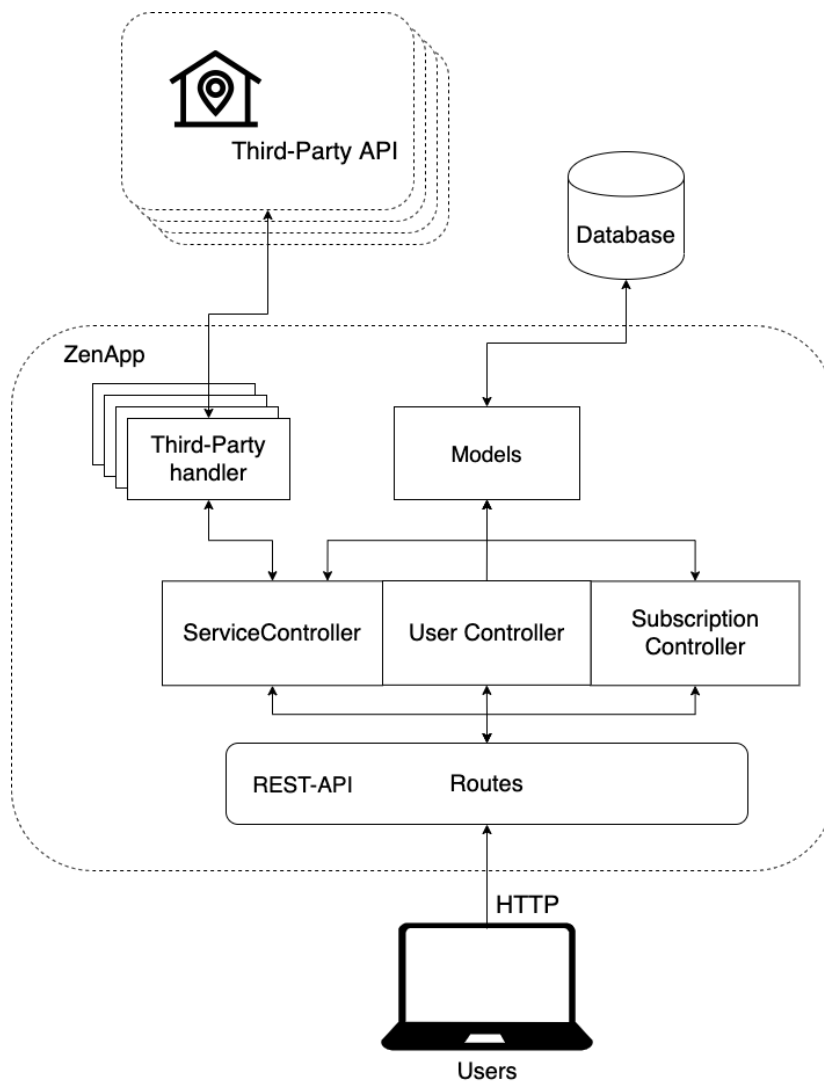


Figure 9 Monolith Architecture, adapted from [37]

The monolith architecture follows the layered architecture pattern[39]. This layered architecture consists of a presentation layer consisting of API-endpoints, a controller layer that contains the business logic, and a data layer that stores persistent data and data models. There is also an external layer which is the multiple third-party services the application will interact with.

In Figure 9 the monolith system architecture is displayed. The presentation layer (REST-API), receives HTTP-request from users through different “*routes*” (Table 1). The routes forward the data received to the correct controller in the controller layer, where the data is processed. The “*User Controller*” is responsible for fetching and creating users and the “*Subscription Controller*” is responsible for subscribing and unsubscribing to different services. The business layer communicates with the data layer through data models that can store and fetch persistent data in a database.

The main feature of the system is the interaction and implementation of multiple third-party services. Each service will have a unique interface for communication that has to be implemented separately. This aspect is handled by the “*Service Controller*” and the “*Third-Party handlers*”. The third-party handlers each presents a standardized interface for interacting with the third-party services. The Service Controller maintains a list of these handlers and is responsible for forwarding requests to the correct handler. Table 1 API Endpoints gives an overview of the API endpoints exposed by the system.

Table 1 API Endpoints

Route	Purpose
/users	Create and fetch users from the database.
/login	Check credentials and return user data.
/subscriptions	Subscribe to services
/services	List available services.
/checksubscriptions	Check service criteria and calculate the distance to the user.

4.2 Serverless Architecture

The serverless architecture is a migration and decomposition of the monolith architecture into serverless functions. Decomposing an existing REST API into serverless functions or building a new API with a serverless approach is a process that has been documented in guides and blog posts[40, 41]. Following these previous examples, the functionality of each endpoint on the monolith was split into its own independent function.

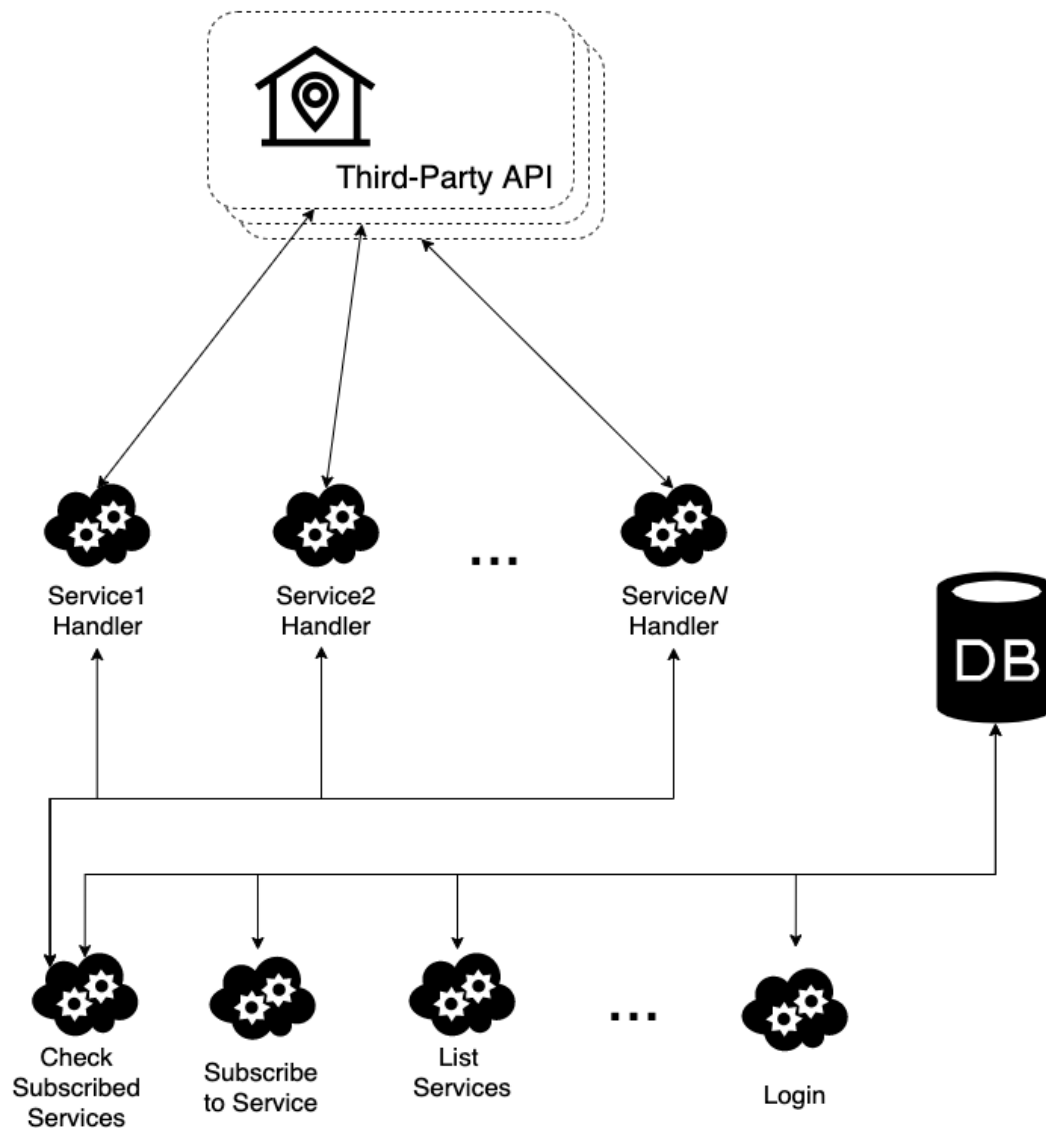


Figure 10 Serverless Architecture

Figure 10 shows an overview of the architecture of the serverless implementation. To deal with the feature of third-party services, a layered

architecture, similar to what is proposed by M. Yan et. al[11] was used. This architecture splits the system into two distinct layers. The first layer consists of functions acting as REST API that clients interact with. It communicates with the database and handles functionality such as fetching users, listing available services, and subscribing to services. The first layer contains equivalent functionality and logic as the monolith, with the exception of the third-party handlers. This first layer communicates with the database and handles functionality such as fetching users, listing available services, and subscribing to services.

The second layer, which is accessed by routing through the “*check subscribed services*” endpoint, consists of a set of serverless microservices that are designed to communicate with external third-party services i.e. the third-party handlers present in the monolith architecture. The goal of this architecture is to promote extensibility by loosely couple the modules that communicate with third parties, meaning services can be added or removed without affecting the overall system.

4.3 Brief Summary

This chapter has covered the high-level design of the two versions of the implemented system and the differences between them. The monolith implementation uses a classic layered architecture, with an API that serves requests from clients, a business layer containing application logic, and a data layer storing persistent data. The serverless design takes the functionality offered by the monolith, splitting it vertically into several independent functions.

Chapter 5 System Implementation

This chapter covers the technical aspect of developing and deploying the monolith and serverless architectures. It also covers the process of selecting frameworks and other components used in the study.

5.1 The Environment of System Implementation

To make the monolith and serverless architectures as comparable as possible, they were implemented in the same programming language, using the same database and deployed on the same cloud platform. This section details the selection criteria and the selected environments and parameters.

Programming Language Criteria – The language should be supported by all major FaaS platforms, to enable replicability on different platforms and industry relevance of the study. Another criterion is the usage in previous serverless research, which can be used to validate and contextualize the findings of this study.

FaaS and Cloud Provider Criteria – The basis for the FaaS provider choice is industry usage as well as previous research. Similar to the language criteria, this is to promote relevance and validity. While usage in previous research is useful for promoting validity, another aspect is the thesis goal of further expanding and broaden the research of serverless. Therefore, the criterion for FaaS provider choice is a balance between these aspects.

Database Selection Criteria – Since this thesis focuses on serverless architectures, it would be appropriate to choose a database solution that does not require any server management. Because of this, the criteria for the database was that it should be a BaaS service.

Table 2 Environment Selection

Programming Language	JavaScript
Language runtime	nodejs10
Cloud Platform	Microsoft Azure
FaaS Platform	Azure Functions
Database	Cosmos DB

In related works and from the informal literature review detailed in Chapter 2, previous research has mainly focused on AWS Lambda and Azure Functions [5, 6, 24]. One limitation of the AWS platform is the AWS gateway, which is used for Lambda functions. The API gateway has a 29-second connection time limit, which means a client's connection is cut off, even if the serverless function has not finished executing[5]. This means it is not possible to measure the actual client response time if it surpasses 29 seconds. To avoid this potential issue, Azure was chosen as the cloud provider. Another aspect is that Azure Functions, being less prevalent than AWS, gives the opportunity to further expand and broaden serverless research on the Azure platform.

JavaScript is available on all major FaaS providers (*AWS Lambda*, *Azure Functions*, and *Google Cloud Functions*). The combination of Azure and JavaScript in previous performance research was also considered when choosing the language. The language runtime was chosen to match both deployments.

For data storage, Azure *Cosmos DB* was selected. This due to being available in the Azure ecosystem and being a backend-as-service database solution.

Table 3 Architectural Properties

	Monolith	Serverless
Code	Single node.js repository	Independent JavaScript functions
Deployment	Azure App Service (PaaS)	Azure Functions App (FaaS)
Idle state	Permanent idle state	No idle state, functions executed when triggered
Resource Allocation	Pre allocated	Allocated on demand
Cost	Static	Dynamic (pay only for used resources)

While the functionality of the implementations is the same, there are some inherent differences in deployment and implementation that differ because of the

distinct architectures, these are showcased in Table 3. The monolith was deployed on *Azure App Service*[42], which is a service on the Azure platform for hosting web applications on a virtual machine. In contrast to the serverless hosting environment, an application hosted on App Service have pre-allocated resources and is “always-on,” even if the application does not receive any traffic.

5.1.1 Azure Functions & Serverless Implementations

As previously mentioned, many FaaS services are implemented differently and tied to a specific cloud provider, this is no different for Azure Functions. With Azure Functions, the primary deployment unit is not individual functions, instead the deployment unit is a *Functions App*[43]. A Functions App contains one or more functions that are scaled and deployed together. The Function App specifies the runtime, which means all functions must be written in the same language. Mixing languages was, however, possible in previous versions [44].

Even though Functions App is the deployment unit, a *function* is the “*primary concept*” of Azure Functions[43]. A function has two components, the code and a configuration file, which among other things specify how the function is triggered. The trigger used for all functions in this study is the HTTP-trigger, which executes a function on an incoming HTTP request. Other triggers include a timed trigger, a database trigger, and more.

Recently, Microsoft introduced the feature of being able to run Azure Functions from a package file[45]. According to Microsoft, this method of deploying functions has the benefits of, in some instances significantly reduce cold starts. It does, however, come with a few limitations. When deploying with a package file, the entire function app becomes *read-only*, meaning it is not possible to edit or create new functions without redeploying the entire Azure Functions application.

These specification details of Azure Functions have architectural implications for the implementation of the serverless system. The atomicity of the Azure Function App raises the interesting question of granularity, is it preferable to keep functions grouped as a *serverless monolith* or keep functions loosely coupled and independent from each other? To explore this, two serverless approaches were considered.

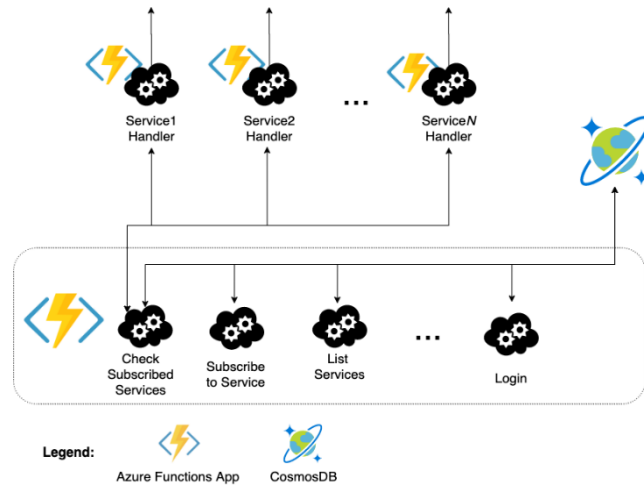


Figure 11 Overview of Serverless Implementation with Azure Functions

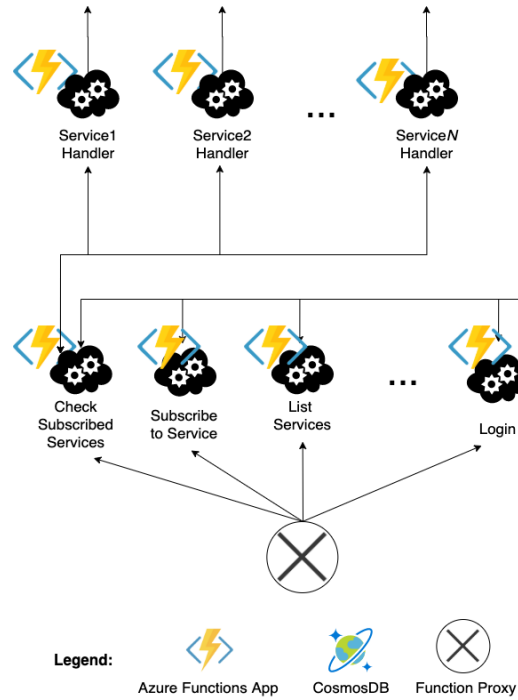
Figure 12 Serverless Microservice Implementation (μ Serverless)

Figure 11 and Figure 12 show the two implementations realized with Azure Functions. As mentioned in Section 4.2, the serverless implementation is separated into two layers. The first layer containing the majority of the application logic, containing the functions corresponding to the monolith REST-API. In the first implementation (Figure 11), these functions were packaged as a single Functions App. The second layer consists of independent functions

handling communication with external services. These are deployed as separate serverless *Azure Functions Applications*. This allows new services to be added and deployed without affecting the deployment of the first layer.

In the serverless microservice implementation shown in Figure 12 (*μServerless*), the application is further separated into self-contained Azure Function Apps, following the microservice pattern of loosely coupled independent services. These Functions Apps are placed behind an *Azure Functions Proxy* which acts as a serverless API gateway and forward incoming request to the appropriate Function App.

5.1.2 Delimitations of Implementation

Since this system is a proof-of-concept, certain features such as security were omitted from the implementations, instead the implementations were focused on the testability of performance, as well as generalizability. For the purpose of evaluating the architectures, the communication with external third-party services API's was not implemented fully in the versions tested. This because it introduces an uncontrolled variable, (a request to a third-party) into the study environment. Instead, the tested systems simulate a third-party service by generating a mock response.

5.2 Architectural Overview

In summary, three distinct implementations, covering four quadrants of the two-dimensional space discussed in Section 2.3, were carried out.

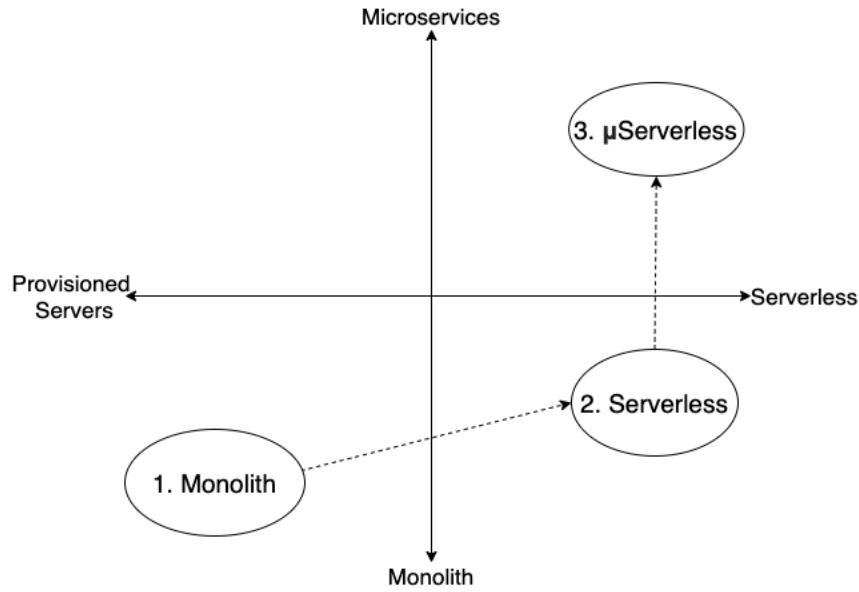


Figure 13 Placement of Studied Architectures

Figure 13 shows the placement of the architectures and the sequence of implementing them. Firstly, the monolith implementation was developed (1). After its completion, the monolith was decomposed into serverless functions. The functions were deployed as a mix of monolith and microservices with the majority of functions grouped as a monolith, using the same code dependencies(2). Finally, the functions were separated into completely decoupled Azure functions apps, routed to through an API proxy(3).

Henceforth these three implementations will be referred to as *Monolith*, *Serverless*, and *μServerless*.

5.3 Key Program Flow Charts

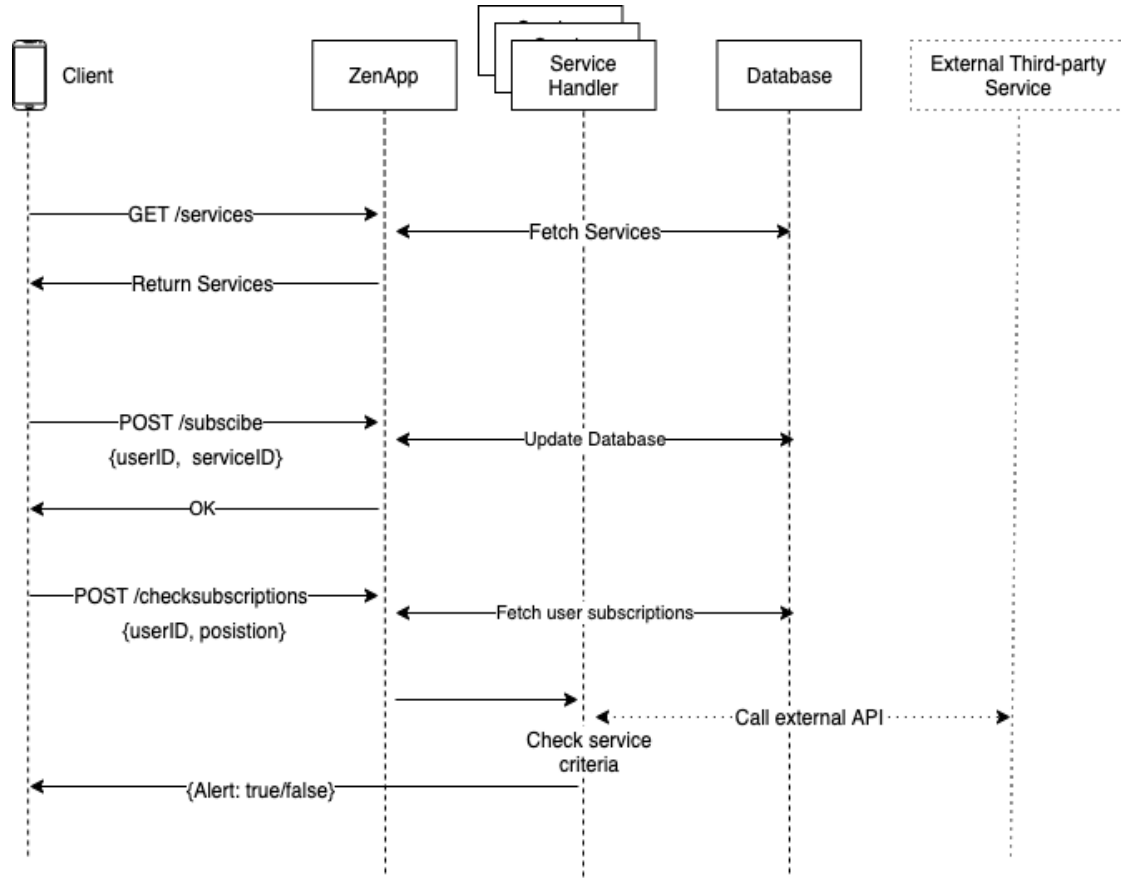


Figure 14 Use case Sequence Diagram

Regardless of the implementation, the Monolith, Serverless, and μ Serverless have the same base functionality. Figure 14 shows a sequence diagram of a simple use case of the system. The sequence diagram showcase the functionality of the proposed system, which is the scenario of a user listing available services, subscribing to a service, and then check the availability of that service. As mentioned in the delimitations in 5.1.2, the dotted lines representing communication with external services are not implemented in the versions evaluated.

Chapter 6 Method

This chapter is split into three sections. The first section covers the hypotheses and goals of the experiments. The second covers the methodology of the experiments and the use case scenarios evaluated. Finally, the third section covers additional data collection and methodology of analysis.

6.1 Hypothesis & Experiment Goal

The goal of this thesis is to explore the implications of building an application in the serverless architecture and to examine the relationships between microservices, monolith, serverless, and PaaS. Section 1.2 presented the research question: RQ1: *What are the effects of implementing the proposed system in a serverless architecture with regards to expected response time?*, and the following sub-questions SQ1, SQ2, and SQ3:

SQ1: How does serverless implementation affect the latency from a user's perspective compared to a monolithic counterpart?

SQ2: What is the impact of cold versus warm starts in a serverless architecture?

SQ3: How does the serverless autoscaling during increased traffic load affect user latency?

These three sub questions needed to be explored by the experiments and served as the basis for the experiment design.

As discussed in Chapter 2, a monolith architecture is a single executable hosted on a web server, while a serverless architecture consists of several independent functions that allocate resources dynamically. One would assume the extra overhead of allocating resources and doing internal communication through the network layer would lead to an increase in response time from the perspective of a client or user. The interesting question, however, especially from a general software industry perspective, is the magnitude in which the latency increases from a monolith to a serverless system. Another aspect is the inherent autoscaling nature of serverless. If a system is under-dimensioned, it is easy to assume that an autoscaling system would be able to handle an increase in traffic better than a non-autoscaling system.

From the perspective of ZenApp, where users are subscribed to services and notified when they are available, latency is not the most critical aspect since requests can be executed passively in the background. However, for features such as logging in, listing available services, etc., responsiveness is still important due to the fact that a user is actively waiting for a response.

As research has shown, users of a web application are less likely to use the application if they perceive an increase in latency[4] and the recommended latency for basic operations is 2 seconds[30] (as discussed in Section 2.5). It is then of interest to see how the latency of the implementations compares to this threshold of 2 seconds. User latency is however affected by a wide range of factors, therefore a static threshold like that is somewhat simplified. It does, however, provide a frame of reference for the experiments. Consequently, the following hypotheses have been constructed:

- H1. The response time of the serverless architectures will be higher than that of the monolith from a client-side perspective in a general case (Non-scaling, not overloaded).
- H2. Cold starts in the serverless architectures will have a negative and noticeable impact on client latency. (>2000ms)
- H3. Due to the autoscaling nature of serverless, during increased load, the serverless architectures will perform better than the monolith and will be able to maintain the 2 seconds threshold.

6.2 Experiments

To be able to confirm the hypotheses and answer the research questions detailed in 6.1, two types of experiments were conducted. The goal of the first experiment (Experiment 1) is to measure the effects of cold starts and thus answer SQ2. The second experiment is meant to answer question SQ3 by measuring the expected response of the architectures during load (Experiment 2). Both experiments covers SQ1 since both experiments measure latency from a user perspective. Table 4 shows an overview of deployments, use case scenarios, and experiments detailed in this section.

Table 4 Experiment Overview

Deployments	Summary
Serverless (Non-Package)	Deployed on Azure Functions as program code. (Not packaged in a zip file)
Serverless (Package)	Deployed on Azure Functions as a packaged zip file.
μ Serverless	A more granular deployment on several Azure Functions apps.
Monolith	Deployed on Azure Web App.
Use case Scenarios	Summary
Scenario 1 – Database Access	Request to the system which fetches an item from the database and returns it to the user.
Scenario 2 - Inter Module/Function communications	Sequence of requests to multiple endpoints of the system.
Experiments	Summary
Experiment 1 Cold starts	Measuring difference between warm and cold starts executions. Answers H1, H2.
Experiment 2 Load Testing	Measuring behavior of tested systems during changing workloads. Answers H1, H3.

6.2.1 Use case Scenarios

Since the goal is to investigate the serverless architecture, not only independent serverless functions, two scenarios reflecting real use cases were selected. While specific to the system implemented in this study, the scenarios are designed to mirror interactions common to general web applications.

Scenario 1 – Database Access – This scenario seeks to investigate the process of accessing several items in a database and returning the results to the

client. In the application, this is translated to a user issuing a request to a system endpoint, and the system returns a list of available services to the user.

Due to this feature having very similar implementations in both of the implementations, another purpose of this scenario is comparing the hosting model of serverless versus PaaS hosting, without being affected by the architectural design pattern.

Scenario 2 – Inter Module/Function communications – While Scenario 1 looks at a trivial use case, Scenario 2 seeks to emulate a non-trivial interaction with a frontend application. This scenario covers a series of sequential requests that simulate a user log in, listing available services, subscribe to a service and then update the current position of the user and check availability of the service.

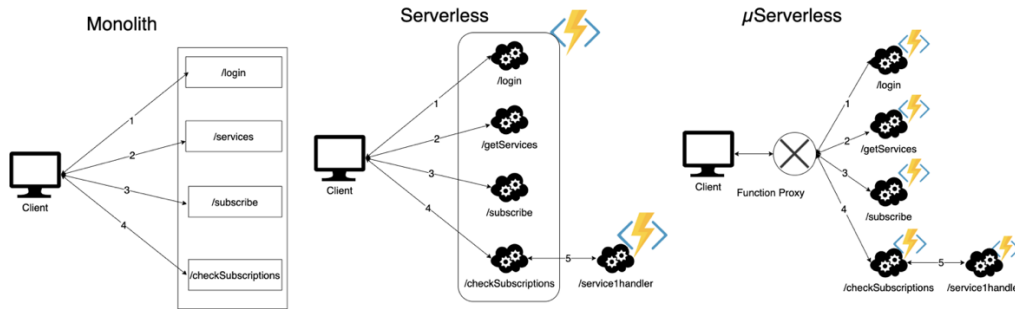


Figure 15 Scenario 2 sequence

Figure 15 shows a visualization of the sequence and API endpoints used in this scenario. For the μServerless architecture, this scenario will involve invocations of functions in multiple apps to carry out a task. In the Serverless system, with the exemption of the *servicehandler* function, all functions are contained in the same app. For the Monolith, the equivalent task is carried out, all logic is however contained inside the application.

6.2.2 Metrics

Client response time (RT) – The time in milliseconds from when the client issues a request to the web application, to the time it receives a response back from the web application.

Scenario Duration – The time in milliseconds for a user to complete the Scenario 2 use case.

95th percentile – The response time where 95 percent of all measurements fall below. This metric gives an estimation of what RT the majority of users will experience while also excluding outliers.

Mean & Median – In addition to the 95th percentile, mean & median values are collected in order to show the spread of measured response times.

Reliability – A request is considered successful if the response contains an HTTP status code 200 (Success). If any other HTTP code is received or the request times out, it is considered unsuccessful. As Kuhlenkamp et al. [26] argues, the ratio of failed and successful request represent an indicator of application reliability.

6.2.3 Experimental Design

Experiment 1 - To measure the impact of cold starts, a similar method to J. Manner et al.[5] was adopted. They claim that most FaaS platforms have shut down the function-container after 20 minutes of idling and the goal of their used method is to trigger a cold start closely followed by a warm start on the same container. This is achieved by sending requests sequentially at regular intervals.

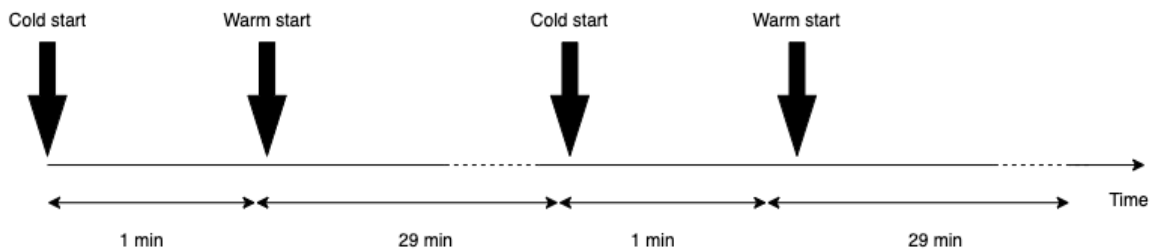


Figure 16 Sequence of cold and warm executions, adapted from [5]

Figure 16 shows the sequence and intervals of requests being sent from the client. In order to get accurate data on user latency, the client records the observed response time from the application. This sequence forces a cold start execution followed by a warm start execution every 30 minutes, assuming no other requests are sent to the serverless function. This experiment was automated and executed with a long-running script detailed in Section 6.2.5. To be able to compare the results to the monolith, the monolith latency was measured in the same script. This generated an equal set of warm, cold and monolith measurements during the same timeframe.

Experiment 2 – Experiment 2 was carried out similarly to the performance tests detailed by Kuhlenkamp et al. [26]. In their study, the researcher stress tests serverless platforms by generating load with simulated users who concurrently makes requests to the application. Since the serverless architecture promises automatic scaling and allocation of resources, it is of interest to see how this compares to the Monolith.

Following the experiment design of Kuhlenkamp et al. the generated workloads were split into three phases, *P0*, *P1*, *P2*. *P0* is a warm-up phase where a constant number of virtual users per second (UPS) send requests to the system under test for 60 seconds. In *P1*, the load is linearly increased for 1 minute. After the scaling phase, *P2* begins and the load stays constant for a duration of 180 seconds. This to see if the platform changes its behavior if under load for a longer period of time. The workloads used in this study are specified in Table 5. The load configurations were determined by a pre-experiment to find suitable values.

Table 5 Experiment 2 Workload configuration, adapted from [26]

	P0	P1	P2
Workload 1	0 UPS	0 -> 60 UPS	60 UPS
Workload 2	0 UPS	0 -> 120 UPS	120 UPS
Workload 3	120 UPS	120 ->400 UPS	400 UPS

6.2.4 Experimental Context & Systems Under Test

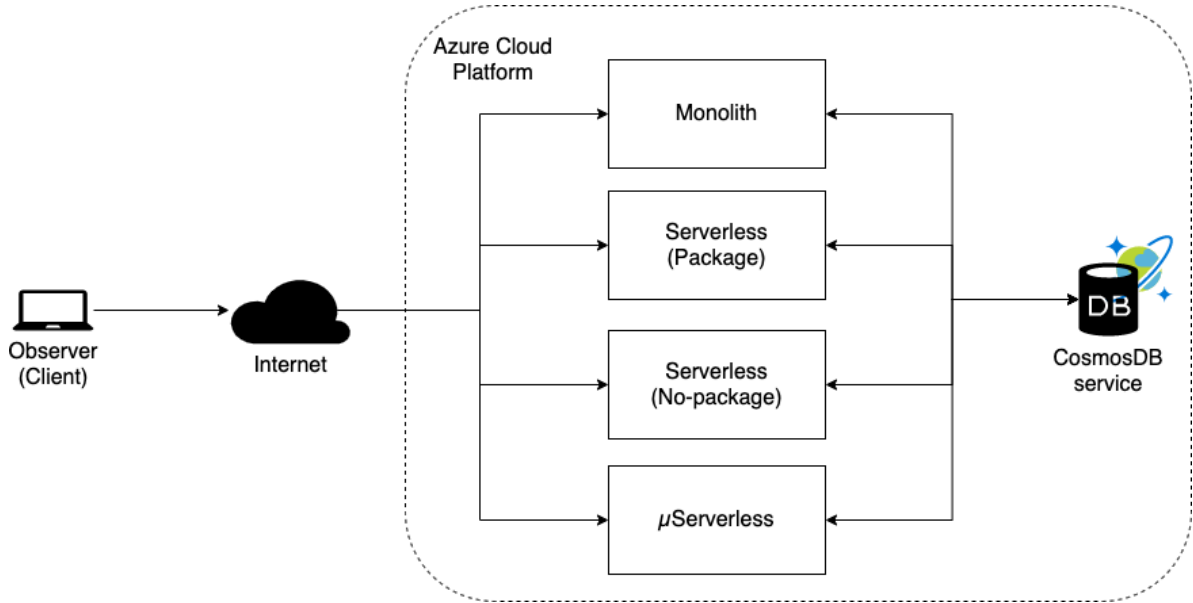


Figure 17 Topology of the experimental environment

The experimental topology is shown in Figure 17. As discussed in 5.1.1, it is possible to deploy Azure Functions as either a package file or as program code. In order to explore the effect of different hosting configurations, two different serverless deployments were tested. Serverless Deployment 1 was deployed as packaged zip files, making the code “read-only” while the other (Serverless Deployment 2) as source code, which makes it possible to make additions, deletions, and updates while the application is running.

The experiments were carried out by the observer (shown in Figure 17). The observer issues request to the applications on the Azure cloud platform and records response times from the applications. This setup simulates a real-world user application issuing HTTP-requests through the internet to a backend application, hosted on the cloud platform. The full tables of specific deployment configurations of *Azure App Service* and the serverless deployments on *Azure Functions* are located in Appendix A.

To mitigate the database being a bottleneck, it was set to the maximum read/write performance setting during the execution of the experiments.

6.2.5 Instrumentation

The load was generated, and data collected with the load testing toolkit *Artillery* described in Section 2.5.1. This tool was chosen because of its simple-to-use and easily scriptable CLI, as well as usage in previous FaaS performance research[26].

```
scenarios:
  - flow:
    - post:
      url : "/login"
      json:
        email : $email
        password : $password
    - get:
      url: "/getServices"
    - post:
      url: "/subscribe"
      json:
        userID : $userID
        serviceObj:
          id: $password
          settings:
            distance: $setting1
            queueTime: $setting2
    - post:
      url: "/checkSubscriptions"
      json:
        userID : $userID
        position :
          long: $long
          lat: $lat
```

Listing 1 Use case scenario configuration

The use case scenarios were configured in an Artillery configuration file as a chain of HTTP requests. Listing 1 shows the scenario configuration. The first request simulates a user login in and authenticating with the system. The second request returns a list of available services. The third request subscribes to a service, and finally, the last request checks the subscribed service by sending a simulated user position to the system.

Experiment 1

```
While True:
    #Record cold start RT
    (timestamp, RT) = run("artillery run serverless_config.yml")
    write("Cold", timestamp , RT)
    sleep(60)
    #Record warm start RT
    (timestamp, RT) = run("artillery run serverless_config.yml")
    write("Warm", timestamp , RT)
    sleep(60)
    #Record Monolith RT
    (timestamp, RT) = run("artillery run monolith_config.yml")
    write("Monolith", timestamp , RT)
    sleep(1620) #Sleep 27 min
```

Listing 2 Experiment 1 pseudo code

For Experiment 1, an automated script was developed. This is a result of response time measurements needed to be collected over an extended period of time.

Listing 2 shows a pseudo-code representation of the used script. The python script uses a loop to measure and record the response times of cold starts, warm start, and the monolith using the Artillery tool. To ensure a cold start in the next loop iteration, the script sleeps for 27 minutes before recording the next triple tuple of results. This means that for every 30 minutes the script is running, one data point for each configuration is collected.

Experiment 2

Experiment 2 was carried out with the Artillery CLI tool and a modified version of a workload generator[46] developed and used by Kuhlenkamp et. al.[26] to record the response time and status code of every sent request. The complete configurations and script used for the experiments have been made available as a GitHub repository[47].

6.2.6 Experimental Execution

All tests were conducted during the period 2020-03-12 to 2020-04-22. In summary, Experiment 1 recorded 160 cold-warm start pairs for Scenario 1 and 169 pairs for Scenario 2. The first experiment covered all four deployments, the Monolith, package and non-package Serverless, and the μ Serverless. In Experiment 2, the non-package Serverless deployment was excluded due to being

deemed superfluous. This because the goal of Experiment 2 is not to explicitly measure cold starts, which in the Azure documentation is the stated difference between package and non-package deployments[45].

6.3 Complementary Observations, Findings & Analysis

Architectural implications are not solely dependent on quantitative performance metrics. To be able to put the acquired results into a broader context and answer the research question regarding architectural implications (RQ2), an analysis of the relationship between different properties or *variables* was carried out. These variables were identified during the informal literature review and relate to the properties of the two architectures. Some additional variables, relating to technical details of the Azure platform, were identified during the implementation phase.

Data were collected both from the experiments as well as the development process (Detailed in Chapter 4 and 5 System Design and System Implementation). The experiments yielded quantitative results, while during the development, complementary unstructured findings and observations were collected. These are observations of both the behaviors and implications of the studied architectures.

The analysis of the data was carried out during and after the data collection process. During this phase, patterns and relationships between variables were studied. This was carried out by triangulating the quantitative data yielded from the experiments, the observational data collected during the implementation and data from previous literature, finding possible connections, relationships, and insights. An overview of this process is shown in Figure 18.

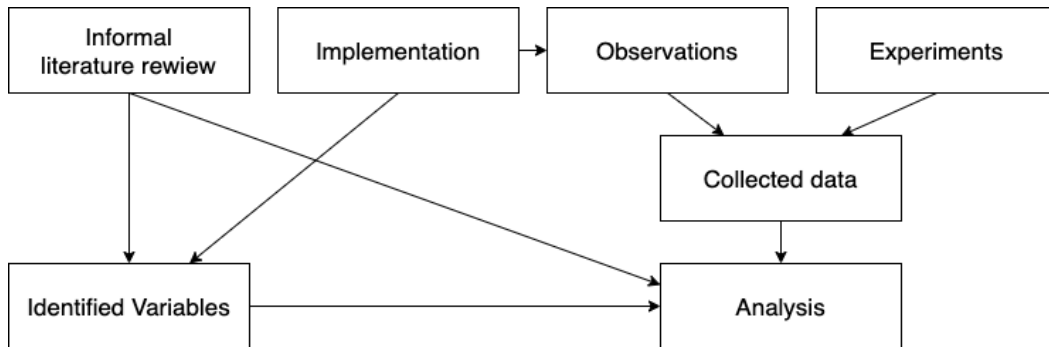


Figure 18 Variable identification and data collection overview

Chapter 7 Results

7.1 Experiment 1 Cold Start Impact

This section covers the result of how cold starts impact user latency. First presented is a compilation of the measured cold and warm start pairs, then a comparison between the two studied deployment methods, followed by a cold start comparison of the Monolith, Serverless, and μ Serverless implementations.

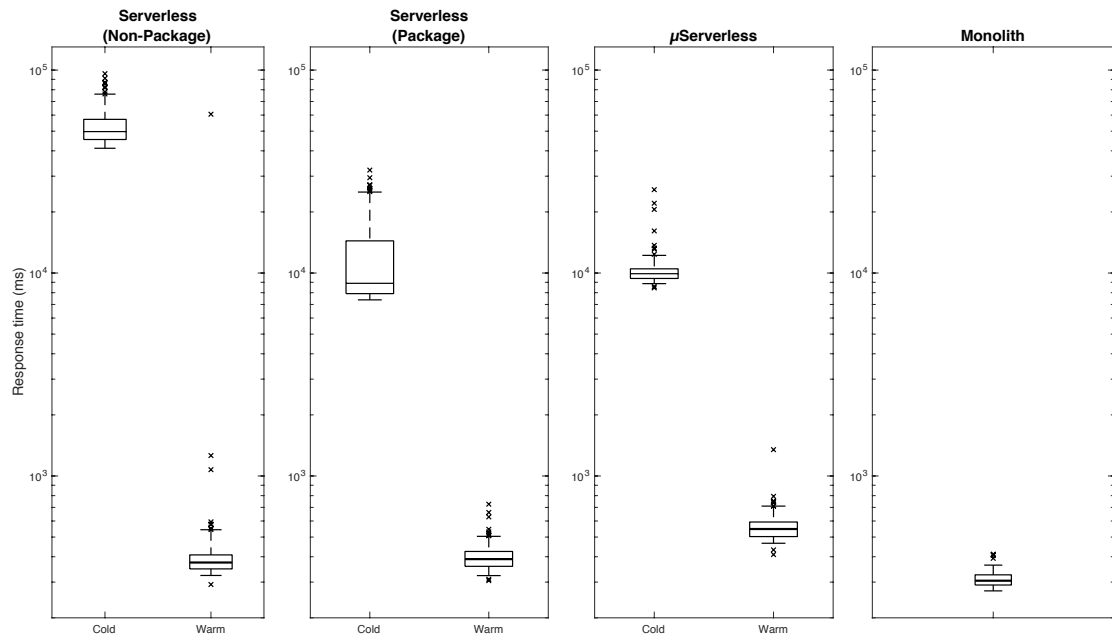


Figure 19 Cold & warm start for Scenario 1

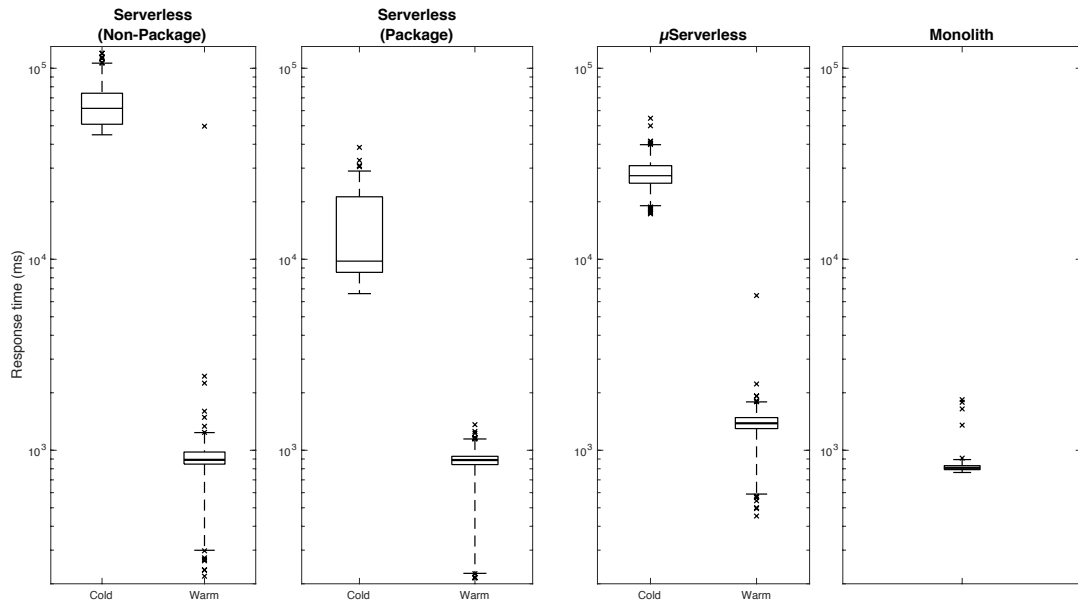


Figure 20 Cold & warm starts for Scenario 2

Figure 19 and Figure 20 shows a box plot of the measured response time for the tested deployments in Scenario 1 and 2. The line crossing each box is the median value while the top and bottom of each box indicate the 75th and 25th percentiles. The dashed line extends to the 5th and 95th percentile and outliers are marked with a cross. For the Monolith, where cold starts are not applicable it is assumed that the application is always warm. There were however no “warmup” requests sent before measurement.

Table 6 Scenario duration of Serverless as Package and Non-Package Deployment

	Cold start (ms)			Warm start (ms)		
Scenario 1	Mean	Median	95 th	Mean	Median	95 th
Non-Package	52663	49754	76112	770	374	543
Package	10308	8897	25053	398	389	504
Scenario 2	Mean	Median	95 th	Mean	Median	95 th
Non-Package	66216	61676	106354	1660	1291	1637
Package	14495	9778	28965	822	889	1146

Table 6 shows that the difference in cold start time between running serverless functions as a package file or non-package, i.e. as uploaded JavaScript

files, is significant. In this case, the mean for the cold starts of the non-package configuration takes just under a minute, while the Serverless app deployed as a package has a mean cold start time of around 10 seconds in Scenario 1.

Table 7 Cold & warm start comparison, Monolith, Serverless, μ Serverless

	Cold start (ms)			Warm start (ms)		
Scenario 1	Mean	Median	95th	Mean	Median	95th
Serverless (Package)	10308	8897	25053	398	389	504
μ Serverless	12194	9922	12216	562	547	710
Monolith	N/A	N/A	N/A	310	304	363
Scenario 2						
Serverless (Package)	14495	9778	28965	822	889	1146
μ Serverless	28365	27392	39810	1370	1385	1791
Monolith	N/A	N/A	N/A	840	809	893

The measured mean, median, and 95th percentile values for cold and warm start scenario durations for the studied architectures are displayed in Table 7. The μ Serverless application generally has a higher response time, both in warm and cold starts.

7.2 Experiment 2 Load Testing

This section covers the results of the load testing experiments, starting with Scenario 1, in which a single endpoint is tested, followed by Scenario 2, which looks at simulated users issuing a sequence of requests. Not all permutations of scenarios, workload, and implementations are covered explicitly. Instead, some interesting data points are highlighted. The complete experiment results can be found in Appendix A.

7.2.1 Scenario 1

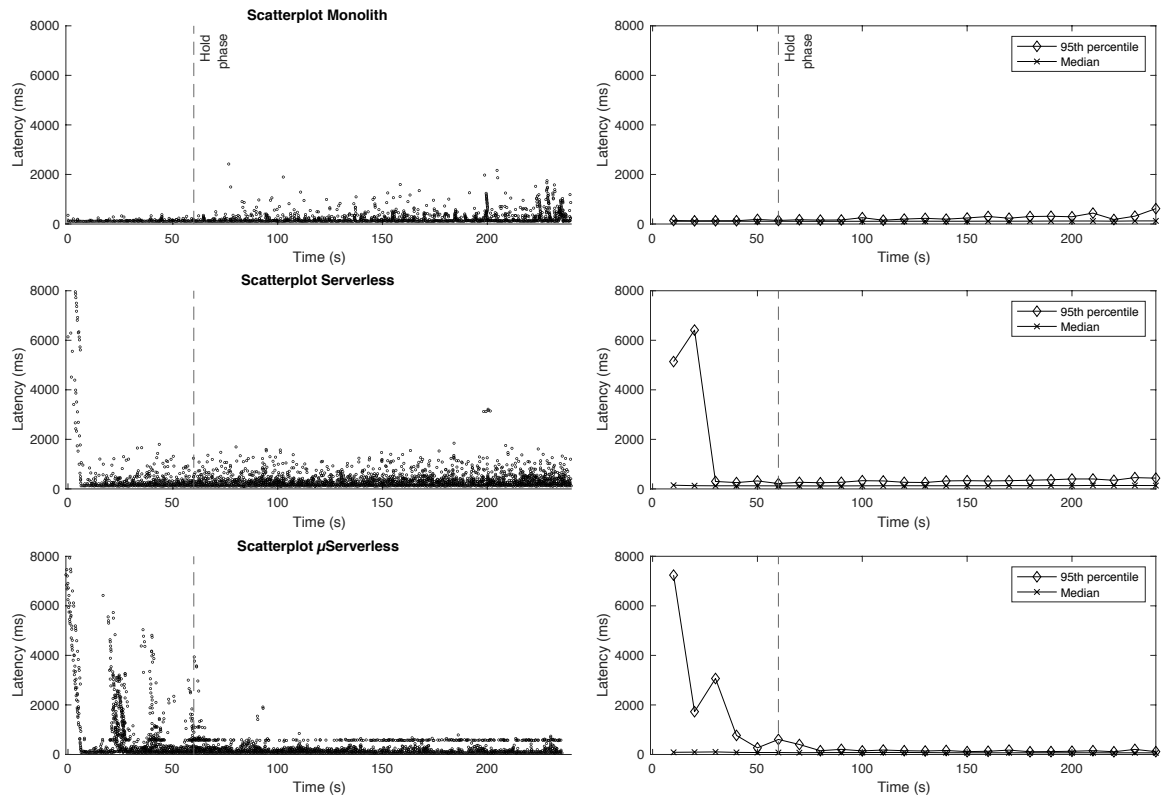


Figure 21 Results of experiment 2, Workload 2, Scenario 1

With the first and second workload, except for the increase in latency due to the cold start in the serverless implementation, there seems to be no significant difference in latency between the FaaS and Monolith implementations. Figure 21 shows the scatterplot, 95th percentile, and median of the studied architectures during workload 2. For this workload the amount of simultaneous request per second linearly ramped up to 120 for 60 seconds and then kept steady for an additional 180 seconds. After stabilizing, the 95th percentile is well below the threshold of 2 seconds. Another noticeable artifact is that the μ Serverless architecture, placed behind a function proxy receives several spikes during the scaling phase that is not observed in the Serverless architecture.

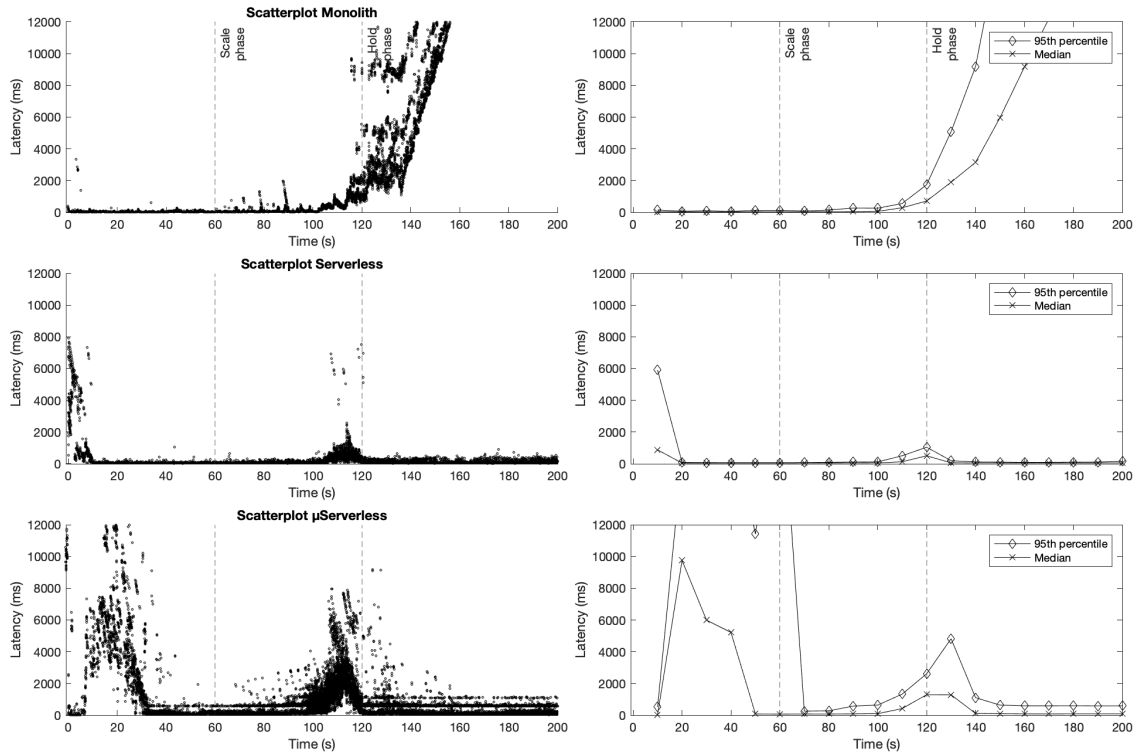


Figure 22 Results of experiment 2, Workload 3, Scenario 1, first 200 seconds

With workload 3, when the number of simulated users are scaled from 120 to 400, different behaviors are observed. Figure 22 shows the first 200 seconds of the experiment. Similar to the previous experiments, the Serverless and μ Serverless implementations initially have a noticeable increase in response time due to cold starts but stabilizes, the Serverless around the 8-second mark, and the μ Serverless much later. When the scaling phase starts and the number of simultaneous starts to increase upwards of 400, all three systems react. Both the Serverless and μ Serverless implementations receive a bump in response time, the μ Serverless, much more severe. After this bump, both the serverless system stabilizes again and remain steady throughout the duration of the experiment. The monolith does not stabilize after the 120-second mark, and instead, the measured response times grow until the experiment is complete. This behavior is further discussed in Chapter 8 *Discussion*. The full plot of all 300 seconds is shown in Appendix A.

Table 8 Aggregate result of Experiment 2, Scenario 1

Workload 1	95th percentile (ms)	Median (ms)	Reliability
Monolith	56	32	100%
Serverless	116	42	100%
μServerless	574	73	100%
Workload 2	95th percentile (ms)	Median (ms)	Reliability
Monolith	251	101	100%
Serverless	304	128.9	100%
μServerless	308	73	100%
Workload 3	95th percentile (ms)	Median (ms)	Reliability
Monolith	87623	30471	97%
Serverless	340	49	100%
μServerless	2473	87	99%

Table 8 shows the 95th percentile, median, and reliability for all workloads during Scenario 1. Note that this table is an aggregate of all requests during a test, meaning the 95th percentile and median value might be significantly higher at a specific point in time, as shown in Figure 21 and Figure 22.

7.2.2 Scenario 2

In Scenario 2, instead of virtual users making single requests, investigates virtual users making a sequence of requests to different endpoints of the system. Therefore, the data presented here are the scenario duration rather than the response time of individual requests. For this scenario, the observer machine generating workload was not able to consistently generate a workload of 400 users per second. Because of this, only the results of workload 1 and 2 are presented.

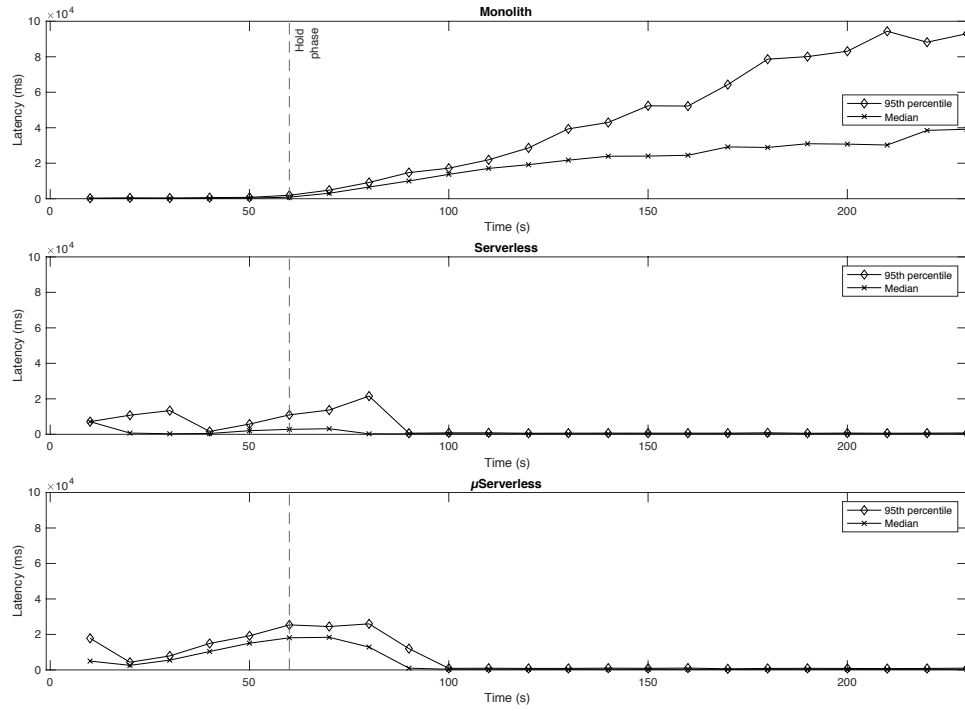


Figure 23 Results of experiment 2, Workload 1, Scenario 2

Figure 23 shows the 95th percentile and median response times of the first workload, where the number of virtual users were scaled from 0 to 60 new users per second (UPS). In comparison to the first scenario, it is apparent that a sequence of request generates a heavier load than a single request. When approaching 60 UPS, the latency of the Monolith steadily increases and is not able to stabilize. Increased latency around the same threshold (60UPS) is also observed in the Serverless and μ Serverless implementations, they both, however, are able to stabilize.

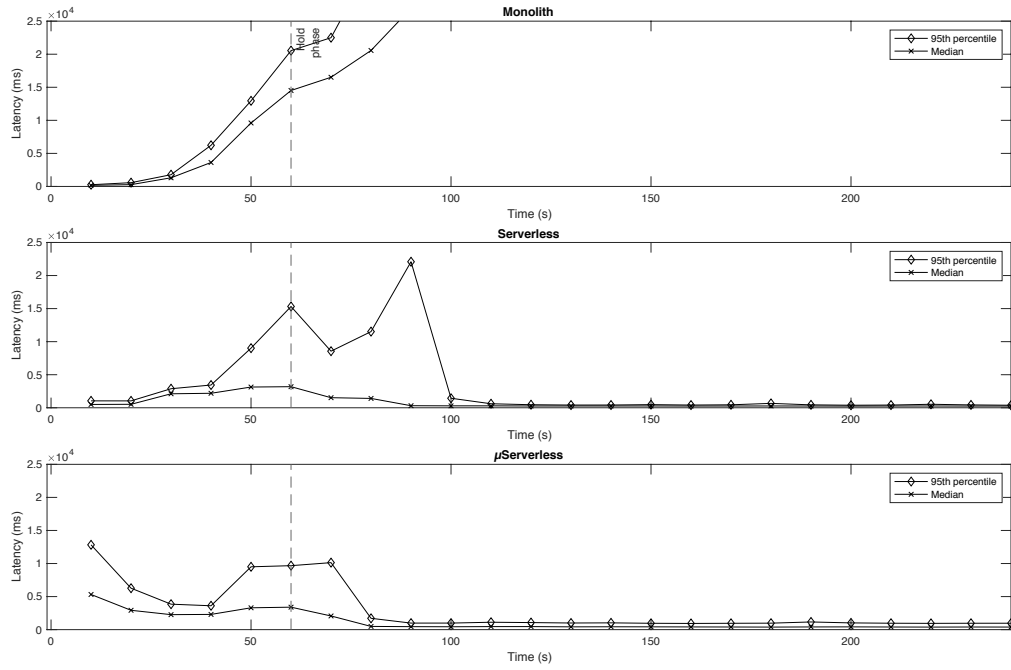


Figure 24 Results of experiment 2, Workload 2, Scenario 2

The same behavior was observed with workload 2 (Figure 24). In order to better showcase the differences between the Serverless and μ Serverless, the y-axis is set to 25000ms. Similar to workload 1, the response times for the Monolith quickly increased while the Serverless and μ Serverless implementations were able to stabilize.

Table 9 Aggregate result of Experiment 2, Scenario 2

Workload 1	95 th percentile (ms)	Median (ms)	Reliability
Monolith	92677	28961	90%
Serverless	3722	263	100%
μ Serverless	19525	413	100%
Workload 2	95 th percentile (ms)	Median (ms)	Reliability
Monolith	342089	271433	79%
Serverless	3680	291	100%
μ Serverless	3951	481	100%

Table 9 shows the overall reliability and measured scenario duration, for workload 1 and 2. The Monolith experienced extremely long response times as well as timeouts, causing a sub 100% reliability.

7.3 Complementary Observations & Findings

During the informal literature review and implementation, variables relating to serverless, and implications of serverless architectures were identified. This section will cover these observed variables and their relations.

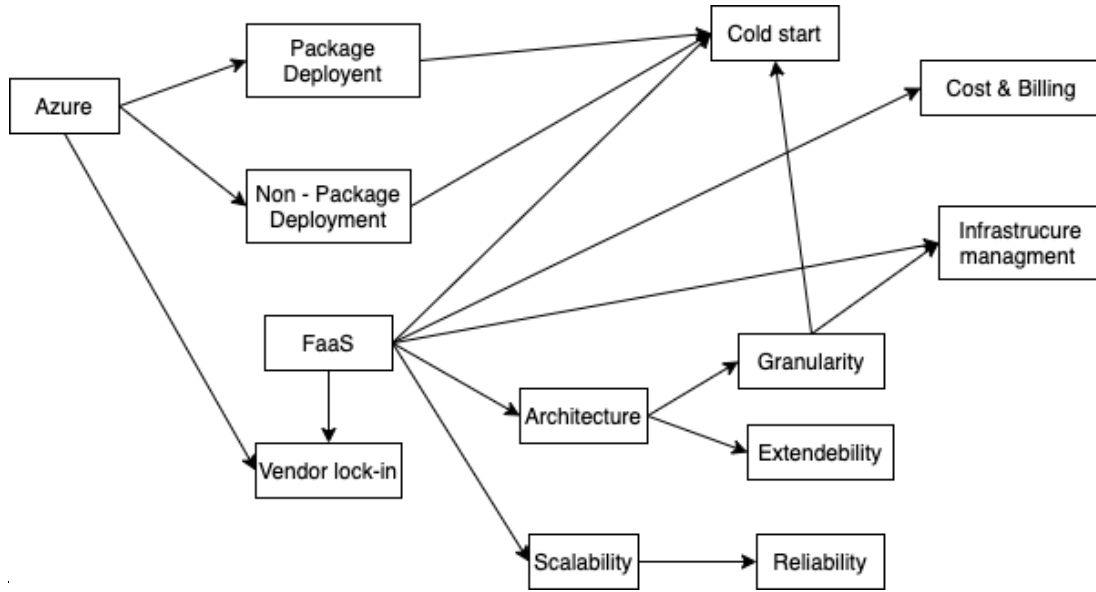


Figure 25 Variable relations

Figure 25 shows a network of relationships among identified variables, based on the results of this study. The purpose of this figure is to serve as a map to show which variables influence one another. The used definition of the identified variables is summarized in Appendix A.

The key takeaways are organized into four major categories and described below. The categories are *Vendor Lock-in*, *Architecture & Extendability*, *Reliability & Infrastructure Management*, and, *Cost & Billing*. The significance of these observations and how they affect serverless systems and applications are further discussed in Chapter 8 *Discussion*.

7.3.1 Vendor Lock-in

While vendor lock-in is not explicitly examined in this study it is often mentioned in literature and is listed as one of the drawbacks of serverless architecture. From observations during the implementation, it is apparent that vendor lock-in is a major trait of FaaS. Even the Serverless framework (described in Section 2.4) which is designed to provide a unified way to build serverless applications, still requires significant refactoring if one decides to migrate from, for example, AWS Lambda to Azure Functions. This is due to the code being structured differently, the different function triggers being offered, integration with other cloud services, etc.. In other words, certain aspects of FaaS platforms are fundamentally different from each other.

7.3.2 Architecture & Extendibility

FaaS heavily influences the architecture of a system since it limits the architecture in certain aspects, e.g. it needs to react to incoming events and carry out short running operations due to the timeout of the cloud provider (10 minutes on Azure). The architecture of the serverless system also affects cold starts. A more granular architecture (μ Serverless) is more severely affected by cold starts.

The architecture also influences extendibility. In the serverless implementations, certain benefits were observed. Additional services could be added to the system simply by registering a new service in the database and configuring a new serverless function, while in the monolith implementation, changing the source code of existing code was necessary. This property, however, seems more correlated with the microservice architectural pattern than directly with *Serverless* or *FaaS*.

7.3.3 Reliability & Infrastructure Management

As the results of Experiment 2 (Section 7.2) indicates, the serverless approach is more reliable than the Monolith. While it is certainly possible to configure a monolith application to auto-scale depending on load, it's not inherent to the architecture, which seems to be the case for the serverless architecture. The monolith is tied to the configurations of the PaaS platform and as a developer, one needs to be concerned about not over or under-provisioning resources. This aspect disappears with a serverless solution since scaling is built

in. Thus, infrastructure management is correlated with FaaS. In terms of managing deployment, no significant difference in terms of effort or degree of difficulty was observed between PaaS and FaaS. In a microservice architecture, several deployments need to be made. Organization and configurations of endpoints and gateways also requires more effort, i.e. the granularity of the system affects the infrastructure management.

7.3.4 Costs & Billing

Table 10 Azure Functions consumption plan, West Europe region, adapted from [48]

	Price	Free per month
Executions	€0.169 per million executions	1 million
Resource Consumption	€0.000014/GB-s	400,000 GB-s

Cost and billing methods are important variables when comparing serverless to traditional PaaS services. The payment plan used for Azure Functions is called a *consumption plan* (Table 10), which follows the serverless standard of only paying for what you use. This plan offers one million monthly executions for free, and the price for executions exceeding that number is €0.169 per million executions. The number of executions is, however, not the only thing that one pays for, *resource consumption* which is the amount of memory a function uses multiplied with the execution time is also included in the cost. The consumption plan offers 400000 Gigabyte seconds for free every month, and additional resource consumption is charged at €0.000014 GB-s. Finally, you also have to pay for the storage of the functions. As an example, using the Azure pricing calculator[49], with the pricing listed in April 2020, the Western Europe region, a serverless app with three million monthly executions, an average of 256 MB memory usage and an average running time of one second would cost €5.14 per month. Adding one gigabyte of storage for the function code would add an additional €0.05 to the monthly cost. In comparison, the basic tier for Azure Web App PaaS service, used in this study costs €11.08 per month. A standard tier recommended for production workloads, and with support for autoscaling would cost €58 per month.

Chapter 8 Discussion

8.1 Performance

Three hypotheses were stated before the experiments measuring the performance of the three architectures.

H1: *The response time of the serverless architectures will be higher than that of the monolith from a client-side perspective in a general case.* Excluding periods of cold starts and scaling, there seems to be no significant difference between the PaaS and FaaS platforms in terms of expected response time. With Experiment 2 (Table 8), WL1 and WL2, the monolith is stable and shows a slightly lower 95th percentile and median latency. Therefore, this hypothesis seems correct in a general, stable case. Accounting for cold starts, the Monolith will generally have shorter response times, as long as the monolith is not overcapacity.

H2: *Cold starts in the serverless architectures will have a negative and noticeable impact on client latency. (>2000ms).* The results show that cold starts are a very noticeable property of serverless. Every tested configuration significantly surpasses the threshold of 2000ms during a cold start. The experiments also show that the severity of cold starts can vary and be reduced. Experiment 1 test two deployment methods, non-package, and package. The ability to be able to edit, add, and remove functions without redeploying might be a benefit. however, this comes at the cost of significantly increased cold start time.

A more granular serverless application will be more severely affected by cold starts. The μ Serverless implementation has longer colds start response times compared to the Serverless implementation in both scenarios. This can be explained by both vertical and horizontal granularity. The μ Serverless implementation is behind a serverless proxy, which also is subjected to colds starts, meaning cold starts will add together and the more depth, i.e. chained serverless functions, will lead to increased cold starts. In Scenario 2, where multiple endpoints are triggered, the μ Serverless implementation requires starting up five separate Function Apps.

The results also indicate that cold starts on the same implementation can vary significantly. For example, the Serverless (Package) has a median cold-start time of 8897ms for Scenario 1 (Table 7) but a 95th percentile time of 25053ms, indicating that cold-start times can be unpredictable and varied.

H3: *Due to the autoscaling nature of serverless, during increased load, the serverless architectures will perform better than the monolith and will be able to maintain the 2 seconds threshold.* The autoscaling nature of serverless allowed the Serverless and μ Serverless to stabilize and fall below the threshold of 95th percentile of response times under 2 seconds. However, it is not able to maintain this throughout the duration of the experiment. The results show a bump in response times when the number of simultaneous requests increases during the scaling phase. This is most likely because of the time to start another instance of the Functions App, i.e. a cold start. As with cold starts, the μ Serverless implementation will receive higher latency during scaling due to its granularity, as shown in Figure 22. First, the Function Proxy is scaled, then the Function behind the proxy is scaled independently, causing a slower stabilization time.

As showcased in Figure 22, the Monolith experiences increasing response times throughout the experiment. While not the main focus of the thesis, it is nonetheless an interesting artifact of the experiments. The reason for this behavior is unknown, however, server-side metrics on the Azure platform show similar response times and corroborate the measured client-side response times. A control experiment was carried out where a second machine located in another network measured response times during the load test. This machine also measured response times akin to those measured by the machine generating load, indicating that the response times described in the results are representative of the systems expected response times. To explore the possibility of the implementation causing this behavior, another control experiment testing a boilerplate API-endpoint without any logic produced similar results, indicating that this behavior stems from the Azure platform. To further strengthen this hypothesis, a control experiment was conducted locally. In this test the behavior of increasing response times was not observed.

One aspect to consider when viewing these results is that the Monolith could be hosted on a more powerful machine that can serve more requests from clients. Despite this, the result captures the inherent differences (automatic scaling, cold

starts) between the “*provisioned servers*” and the “*serverless*” end of the spectrum.

Comparing these results to the informal literature review and related work, the results seem to mostly align. Albuquerque et al.[10], while using Amazons services, also finds that a PaaS platform performance is equivalent to the performance of FaaS, but similarly, that cold starts can significantly impact the performance. In terms of cold starts, the results indicate response times in the same range (10 seconds) as measured by J Manner et al.[5] for JavaScript functions on Azure.

One noticeable divergence from other research is that compared to Kuhlenkamp et al.[26] who uses a similar load generation method, measures significantly more failed requests (95%) on the Azure Functions platform. Their deployed Function App is not able to stabilize after the 300 seconds and deems it unsuitable for applications with volatile workloads. This is not the case for the experiments of this study, where the serverless implementations are able to stabilize for all tested workloads. One potential reason for this might be because of the difference in function implementation, while also using Node.js version 10 runtime, they instead of interacting with a database, calculate whether or not a number is a prime, which is potentially more memory and CPU intensive. It could also simply be that the platform itself has changed since the study, this possibility is further discussed in Section 8.5 *Threats to validity and reliability*.

8.2 Architectural implications of Serverless

An application built with serverless technology will have certain characteristics. One of these is cold starts. Experiments show that cold starts of around 10 seconds or more are to be expected. This is something that has to be taken into account. A system that needs near-instantaneous feedback such as autonomous vehicles might not be a suitable application for serverless, especially if the usage is irregular and cold starts are expected to occur frequently.

Another characteristic of Serverless is the very tight coupling to the FaaS platform. This has been described as vendor lock-in. It is apparent that one chooses the FaaS platform then develops a serverless application, rather than the other way around, developing a serverless application and then choosing a provider to host it. Because of this, vendor lock-in might not be the most

appropriate term to describe this tight coupling. Instead, a more appropriate viewpoint would be to view each FaaS platform like a unique service and tool, and applications are built for that specific platform, not a general application that can be deployed anywhere.

Serverless and FaaS impose some inherent limitations on a system. Functions have a limited running time, meaning functions that carry out heavy calculations or wait operations needs to be ensured not to exceed this limit. Variables are not guaranteed to persist between invocations of the functions meaning application needs to be stateless or store state externally. Since the serverless applications do not idle, the FaaS platforms enforce an event-based design, limiting possible applications. To build a system with a serverless architecture, one needs to adhere to and consider each of these limitations.

Perhaps the most emblematic characteristic, from which the term *serverless* comes, is the abstraction of server infrastructure. As the results imply, reliability increases in a serverless architecture due to the dynamic allocation of resources. This means that serverless applications will be able to handle unexpected growth and spikes in traffic that a non-serverless application might not be able to sustain, like the Monolith in Experiment 2. A developer creating a serverless application can be confident that the code deployed will be able to run during any load. With a dedicated server, a developer needs to deal with the added complexity of predicting application workload and if the application userbase grows, handle and configure scaling and load balancing.

8.3 Pricing & Cost

One of the selling points of serverless is the monetary cost benefits. If a system is not being used at the moment, you do not have to pay for it. Azure also offers a seemingly generous free tier. One million execution provided for free every month means a Functions App can be called 22 times every minute for free. As presented in Section 7.3.4, an application that receives millions of requests and has relatively long-running and memory intense functions can be significantly cheaper than the standard and even the basic tier of Azure Web App. This implies that Azure Functions and serverless architecture in general, are an economically sound choice when developing and deploying an application.

Azure Functions are, however, not necessarily always the most optimal choice from an economic perspective. Experiment 2 (Section 7.2) shows that the monolith can handle a sustained 60 requests per second and still fall below the threshold of a 95th percentile latency of 2 seconds. Assuming the load stays constant and assuming the monolith can handle this load indefinitely, the monolith implementation serves 3600 requests every minute. The amount of handled request per month would then be $3600 * \text{minutes per month} (60 * 24 * 30) \approx 155,5 \text{ million request per month}$. I.e. the monolith can handle 155.5 million requests for a monthly cost of €11.08.

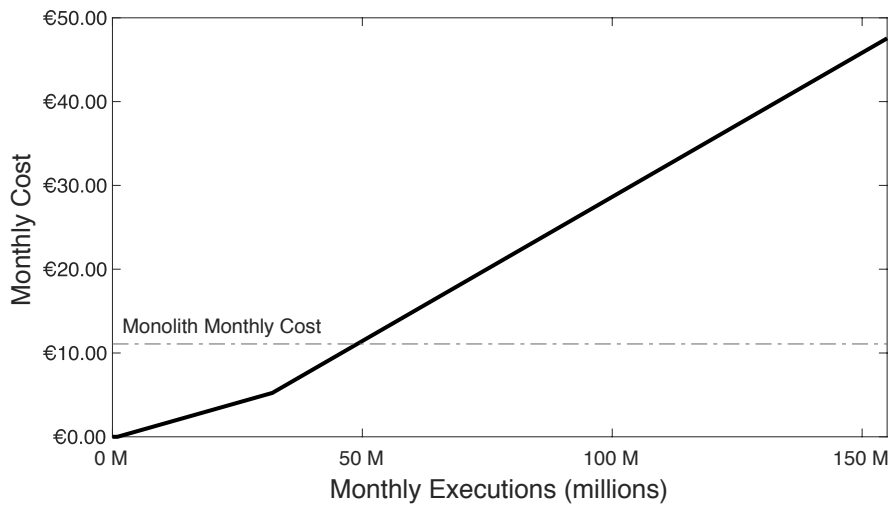


Figure 26 Azure Functions cost calculation example.

Figure 26 shows the monthly cost of hosting the monolith and the calculated price for the serverless implementations per monthly executions. Prices were calculated with the minimum average memory usage of 128MB and the minimum execution time of 100ms (median value for Serverless latency in Workload 1, Scenario 1 was 42). In the figure, the cost of the serverless implementations surpasses the cost of the monolith at 50 million monthly executions. Using the previous example of the monolith serving 155.5 million requests per month, calculating the same number of executions for the serverless implementations gives a monthly cost of €48, significantly more than the monolith hosting cost. What this example shows is that while serverless can be a cheap solution, there exist certain scenarios where serverless can be significantly more expensive than a PaaS option.

To summarize, an application that has occasional or unpredictable traffic has the potential to reduce its infrastructure cost by being developed with a serverless architecture. If the traffic can be predicted, one should consider if the dynamic cost of serverless exceeds the static costs of provisioning servers for the system.

8.4 Comparison of Monolith, Serverless & μ Serverless

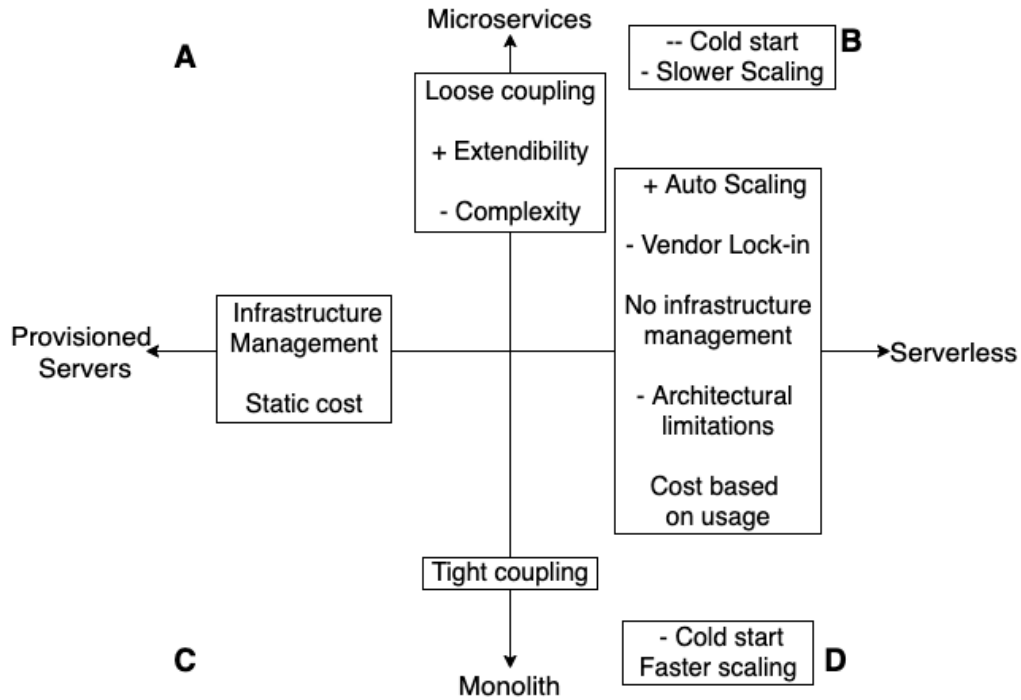


Figure 27 Mapping of traits

Figure 27 shows a mapping of observed characteristics to the architectural plane. Traits that are generally considered negative is marked with “-” while positive is marked with “+”.

From the perspective of ZenApp, based on the results from this study, there are cases to made for choosing either a Serverless or Monolith approach. The Monolith, placed in the *C* quadrant in Figure 27, was straightforward to develop and deploy. The main drawback is extensibility. Due to the nature of the application, external services need to be added and removed. Due to the Monolith being a single executable, this cannot be achieved without modifying and redeploying the entire application. Another benefit is not being affected by cold starts which can impact users of web applications, potentially reducing the usage of the application due to long wait times. While this is certainly more of a

concern for applications with a low number of users, even applications with a large userbase can be subjected to cold starts due to automatic scaling during traffic spikes or periods of low traffic, e.g. nighttime.

The Serverless and μ Serverless approach do allow dynamically adding and removing of third-party services, this trait does, however, correlate more to the A , B axis of microservices more so than serverless. The Serverless (D) and μ Serverless (B) are affected by the limitations discussed in Section 8.2, meaning ZenApp's design is limited and tied to the Serverless platform, but also receive the inherent benefits of serverless such as autoscaling, no infrastructure management, and cost based on usage.

The more granular μ Serverless approach (quadrant B) introduce more loose coupling at the cost of longer response times and cold starts. A more loosely coupled system is also more complex to develop and deploy. Therefore, it would be appropriate to investigate which parts should be split into independent services and which should be grouped together. In the case of ZenApp, a very granular approach such as in the μ Serverless architecture does not seem like the correct approach.

Another option not explicitly investigated is a combination of the monolith and serverless approaches, placing in the center of Figure 27. The basic functionality such as handling users, listing services, etc. could be contained in a single always-on web application, the third-party service handlers, however, developed as separate serverless applications. This approach would remove cold starts affecting user interaction with the system since users do not interact directly with the services but still allow services to be added and removed dynamically without significantly affecting any other part of the system.

In conclusion, there are trade-offs and implications for choosing either of the architectures. An important take-away is that the novel serverless architecture is an appropriate alternative to traditional architectures.

8.5 Threats to Validity and Reliability

This empirical study aims to follow the guidelines proposed by Kitchenham et al.[34] with the goal of making the methodology and experiments as transparent and repeatable as possible. There are, however, some factors that might affect the overall validity and reliability of this study. These factors are

separated into construct validity, internal validity, external validity, and reliability and addressed below.

8.5.1 Construct Validity

The setup of this study intended to evaluate the architectural implications of choosing a serverless versus a monolith architecture for a proposed system. This has been done by developing three separate systems and comparing the different implementations in terms of performance. With controlled experiments in software research, there is a risk that the technique that is evaluated is too oversimplified. When comparing implementations, there is a balance between simplicity and complexity. On one hand, rudimentary implementations, where all interfering variables and factors have been removed or minimized, are more comparable. Such comparisons have little meaning in a real-world scenario. Non-trivial implementations, on the other hand, might possibly be so radically different that comparisons become difficult since they are affected by a wide range of possible factors, such as developer skill and design choices. In order to balance this, the architectures that are examined in this paper are proof-of-concept implementation, and while simplified, still contain logic and design that a real implementation could utilize.

8.5.2 Internal Validity

The implementations of the architectures being evaluated in the experiments are subjected to some inherent bias due to both being developed by the author. The author's skill in terms of implementing the proof-of-concept application is also a factor threatening validity. The way one architecture is implemented could give significant benefits or disadvantages during an evaluation, therefore the conclusions of this could have some inherent biases.

Measuring and collecting response times from cold starts and warm starts take a significant amount of time (30 minutes per cold-warm start pair). This leads to the relatively small sample size of 160 for Scenario 1 and 169 for Scenario 2, which could harm the validity of the results. Response times are also affected by a wide variety of external factors such as the physical location of servers, network traffic, etc. While this thesis presents client-side response times, in order to ensure that the observer does not produce skewed results, the results

were also compared to the server-side performance statistics on the Azure platform.

8.5.3 External Validity

To promote the external validity of the results and conclusions, the use case scenarios evaluated were specifically chosen for their applicability to general web applications, not specifically relevant to the implemented application.

The cloud platform itself can be a factor affecting validity. Due to the new and evolving technology as well as the black-box nature of serverless and FaaS, the validity result of the experiment might be affected by future updates or modifications to the Azure Functions platform.

Finally, as previously mentioned, there exist many previous studies on the performance of FaaS, serverless architectures, and cold starts. In order to strengthen the external validity of this study, the produced results are compared to previous research. Triangulation of previous research and scientific literature is also used together with the produced result in order to draw the conclusions of this study.

8.5.4 Reliability

To promote the reproducibility of the results, all source code of the system itself and scripts used for testing are made available as open-source on GitHub[47]. The goal has been making the experiment methodology and configurations of the cloud platforms as transparent as possible. Reliability, like validity, might also be affected by future updates to the Azure cloud platform.

8.6 Work in a Wider Context

From an ethical point of view, the tight coupling to a specific provider i.e. vendor lock-in might be problematic as it potentially creates barriers to entry for competing serverless providers and disincentivizes switching platforms due to substantial switching costs. Terms of services might change over time, but due to the difficulty of switching providers, customers will be forced to stay with unfavorable terms. From a broader perspective, this might be cause for some concern and discussion for serverless and cloud computing as a whole.

Another and perhaps one of the most interesting aspects of serverless, especially from a societal and ethical perspective, is green computing. Migrating systems to a serverless model could dramatically lower energy consumption and optimize resources in today's data centers. Forbes wrote an article in 2015[50] claiming that 30 percent of servers in data centers all over the world are in a "comatose" state. The article also claims that on average, a typical server is using five to 15 percent of its maximum computing output per year. One solution to this inefficiency could be serverless computing, where instead resources would be allocated where needed instead of being constantly reserved for idle servers. Then the responsibility of minimizing environmental damage would be in the hands of the serverless cloud providers, not individual companies or developers, which is arguable in a better position to do so since they have the ability to make decisions and direct resources on a large scale.

What this study has shown, is that serverless architecture is a valid approach when developing software. If a system meets the necessary requirements, such as it can be built as a collection of short-running, stateless functions there are few downsides to taking the serverless approach. Serverless computing technology can be expected to continue to grow, develop, and become even more widespread in the future.

Conclusion

This study has explored and showcased different characteristics and traits of different software architectures in the spectrum of microservices, monolith, dedicated servers, and serverless. It has also showcased the performance, scalability, and applicability of Microsoft Azure Functions. Chapter 1 presented two research questions (RQ1, 2) with the purpose of exploring the implications of serverless architectures, both in the context of the ZenApp proof-of-concept system as well as in a more broad, general case. Based on experiments, observations, analysis, previous research, and discussion, the following conclusions can be stated:

RQ1: *What are the effects of implementing the proposed system in a serverless architecture with regards to expected response time?* Along with this question, three sub-questions (SQ1,2,3) were asked. Using the study's findings, each question is answered below:

SQ1: *How does serverless implementation affect the latency from a user's perspective compared to a monolithic counterpart?* In a general scenario (warm start, no simultaneous scaling) a serverless implementation will not have any significant difference in user-perceived latency compared to a monolith implementation, however, the monolith performed slightly better in terms of latency.

SQ2: *What is the impact of cold versus warm starts in a serverless architecture?* Cold starts are an inherent characteristic of serverless architecture that has a severe impact on response times. A cold start can give response times of several seconds and in some scenarios upwards of one minute, while warm starts give response times in the range of 10s or 100s of milliseconds. A more granular architecture approach (microservices) will also be more severely affected by cold starts.

SQ3: *How does the serverless autoscaling during increased traffic load affect user latency?* The serverless implementations were able to handle all tested workloads and stabilize with acceptable user response times. During increases in load and before stabilization, some periods of longer response times are expected.

RQ2: *What are the observed implications of choosing a serverless architecture to fulfill the requirements of the system?* By building the proposed system with a serverless architecture, it will have certain characteristics. Beyond the performance and autoscaling traits already mentioned with RQ1, a serverless system will be tightly coupled to a cloud platform and be subjected to limitations that a non-serverless system would not, such as being required to be event-based and store stateful variables externally. While serverless applications can be developed with different levels of granularity, they promote a style of architecture of independent functions. Functions can then be easily added and removed without affecting any other part of the system. Using a more granular approach allowed additions to the system without affecting the overall application, something not possible in a monolith system. The ability to pay for resource consumption has the potential to lower infrastructure costs compared to a PaaS service.

Future Work

Finally, this section presents some suggestions for further avenues of research on the topic of FaaS and serverless architectures. With this study's focus on a proof-of-concept system, the natural progression would be to explore and evaluate a full-fledged serverless system, with complete functionality, user authentication, and security. This to further explore serverless architecture and FaaS in an industry context.

Another avenue is the tight coupling to the cloud provider. With vendor lock-in being a frequently discussed topic regarding serverless technology, it is of interest to investigate what consequences a platform choice has on a system. Thus, a further categorization and mapping of different FaaS providers and what implications for a serverless architecture come with the platform choice could serve as a useful contribution to the topic.

References

- [1] Adzic, G. and R. Chatley. *Serverless computing: economic and architectural impact*. in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017.
- [2] Baldini, I., et al., *Serverless computing: Current trends and open problems*, in *Research Advances in Cloud Computing*. 2017, Springer. p. 1-20.
- [3] Roberts, M. *Serverless Architectures*. 2018 [2020-01-30]; Available from: <https://martinfowler.com/articles/serverless.html>.
- [4] Arapakis, I., X. Bai, and B.B. Cambazoglu. *Impact of response latency on user behavior in web search*. in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. 2014.
- [5] Manner, J., et al. *Cold start influencing factors in function as a service*. in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018. IEEE.
- [6] Jackson, D. and G. Clynh. *An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions*. in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018. IEEE.
- [7] Van Eyk, E., et al. *The SPEC cloud group's research vision on FaaS and serverless architectures*. in *Proceedings of the 2nd International Workshop on Serverless Computing*. 2017.
- [8] Hellerstein, J.M., et al., *Serverless computing: One step forward, two steps back*. arXiv preprint arXiv:1812.03651, 2018.
- [9] Villamizar, M., et al., *Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures*. *Service Oriented Computing and Applications*, 2017. **11**(2): p. 233-247.
- [10] Albuquerque Jr, L.F., et al. *Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS*. in *ICSEA*. 2017.
- [11] Yan, M., et al. *Building a chatbot with serverless computing*. in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. 2016.

- [12] Seaman, C.B. and V.R. Basili, *Communication and organization: An empirical study of discussion in inspection meetings*. IEEE Transactions on Software Engineering, 1998. **24**(7): p. 559-572.
- [13] Lewis, J. and M. Fowler. *Microservices, a definition of this new architectural term*. 2014 2020-01-30]; Available from: <https://martinfowler.com/articles/microservices.html>.
- [14] Richardson, C., *Microservices Patterns: With Examples in Java*. 2019: Manning Publications.
- [15] Thönes, J., *Microservices*. IEEE software, 2015. **32**(1): p. 116-116.
- [16] Villamizar, M., et al. *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud*. in *2015 10th Computing Colombian Conference (10CCC)*. 2015. IEEE.
- [17] Ihde, S. and K. Parikh, *From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture*. 2015.
- [18] Calçado, P. *Building Products at SoundCloud —Part I: Dealing with the Monolith*. 2014 2020-02-03].
- [19] Hendrickson, S., et al. *Serverless computation with openlambda*. in *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. 2016.
- [20] Roberts, M. and J. Chapin, *What is serverless: understand the latest advances in cloud and service-based architecture* 2017, O'Reilly Media.
- [21] *What is function as a service?* [cited 2020 2020-04-25]; Available from: <https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/>.
- [22] Eivy, A., *Be Wary of the Economics of" Serverless" Cloud Computing*. IEEE Cloud Computing, 2017. **4**(2): p. 6-12.
- [23] Tresness, C. *Understanding serverless cold start*. 2018; Available from: <https://azure.microsoft.com/sv-se/blog/understanding-serverless-cold-start/>.
- [24] Lynn, T., et al. *A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms*. in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2017. IEEE.
- [25] *Serverless Framework*. 2015 2020-02-17]; Available from: serverless.com.

- [26] Kuhlenkamp, J., et al., *Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-driven Design of Serverless Applications*. 2020.
- [27] Kuhlenkamp, J. and S. Werner. *Benchmarking FaaS Platforms: Call for Community Participation*. in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018. IEEE.
- [28] Van Eyk, E., et al. *A SPEC RG cloud group's vision on the performance challenges of FaaS cloud architectures*. in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018.
- [29] Herbst, N.R., S. Kounev, and R. Reussner. *Elasticity in cloud computing: What it is, and what it is not*. in *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*. 2013.
- [30] Rempel, G. *Defining standards for web page performance in business applications*. in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. 2015.
- [31] *Artillery.io*. 2020-02-05]; Available from: <https://artillery.io/>.
- [32] Grambow, M., et al. *Benchmarking Microservice Performance: A Pattern-based Approach*. in *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20)*. 2020.
- [33] *Apache JMeter*. 2020 2020-02-25]; Available from: <https://jmeter.apache.org/>.
- [34] Kitchenham, B.A., et al., *Preliminary guidelines for empirical research in software engineering*. IEEE Transactions on software engineering, 2002. **28**(8): p. 721-734.
- [35] Kitchenham, B., *Empirical paradigm—the role of experiments*, in *Empirical Software Engineering Issues. Critical Assessment and Future Directions*. 2007, Springer. p. 25-32.
- [36] Runeson, P. and M. Höst, *Guidelines for conducting and reporting case study research in software engineering*. Empirical software engineering, 2009. **14**(2): p. 131.
- [37] Andell, O. and J. Holmström. *Shared Requirements Zenon Application*. 2020 [cited 2020 February, 17th]; Available from: <https://github.com/OAndell/Masterthesis-shared-repo>.
- [38] Holmström, J., *Distributed Queries - An Evaluation of the Microservice Architecture*. 2020.

- [39] Richards, M., *Software architecture patterns*. Vol. 4. 2015.
- [40] Holland, B., *How to migrate a traditional Express API to Serverless and save tons of money*. 2018: dev.to.
- [41] *Build Nodejs APIs using Serverless*. 2018 [cited 2020 2020-03-04]; Available from: <https://azure.microsoft.com/en-us/resources/videos/build-nodejs-apis-using-serverless/>.
- [42] *Azure App Service overview*. 2017 2017-01-04 2020-02-25].
- [43] *Azure Functions developers guide*. 2017 [cited 2020 2020-03-17]; Available from: <https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference>.
- [44] *Azure Functions runtime versions overview*. 2019 [cited 2020 2020-03-17]; Available from: <https://docs.microsoft.com/en-us/azure/azure-functions/functions-versions>.
- [45] *Run your Azure Functions from a package file*. 2019 [cited 2020 2020-03-09]; Available from: <https://docs.microsoft.com/sv-se/azure/azure-functions/run-functions-from-deployment-package>.
- [46] Werner, S. *FaaS Elasticity Engineering*. 2020 [cited 2020 2020-04-01]; Available from: <https://github.com/tawalaya/FaaS Elasticity Engineering/tree/master/workloadGenerator>.
- [47] Andell, O., *Thesis Github repository*. 2020.
- [48] *Azure Functions pricing*. 2020 [cited 2020 05-04]; Available from: <https://azure.microsoft.com/en-us/pricing/details/functions/>.
- [49] *Azure Pricing calculator*. [cited 2020 2020-04-06]; Available from: <https://azure.microsoft.com/en-us/pricing/calculator/>.
- [50] Kepes, B., *30% Of Servers Are Sitting "Comatose" According To Research*, in *Forbes*. 2015.

Appendix A

A.1 Deployment configuration

Table 11 Monolith Deployment Configuration

Service	Web App
Publish	Code
Runtime Stack	Node.js10.x
Operating System	Linux
Region	West Europe
Sku and Size	B1

Table 12 Serverless Deployment (Package) Configuration

Service	Functions App(s)
Runtime Stack	Node.js10.x
Region	West Europe
Run from Package	Yes

Table 13 Serverless Deployment (Non-Package) Configuration

Service	Functions App(s)
Runtime Stack	Node.js10.x
Region	West Europe
Run from Package	No

Table 14 μ Serverless Deployment Configuration

Service	Functions App(s)
Runtime Stack	Node.js10.x
Region	West Europe
Run from Package	Yes

A.2 Experiment 2, full results

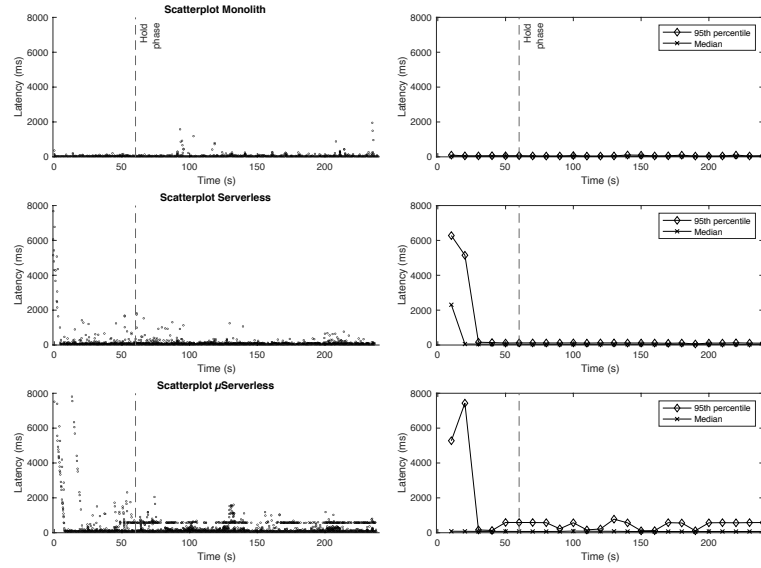


Figure 28 Scenario 1, Workload 1

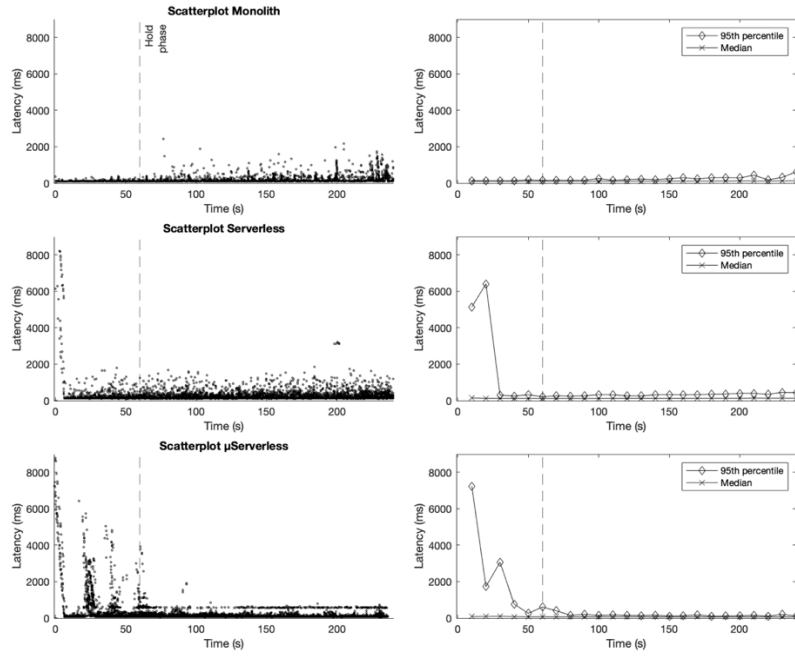


Figure 29 Scenario 1, Workload 2

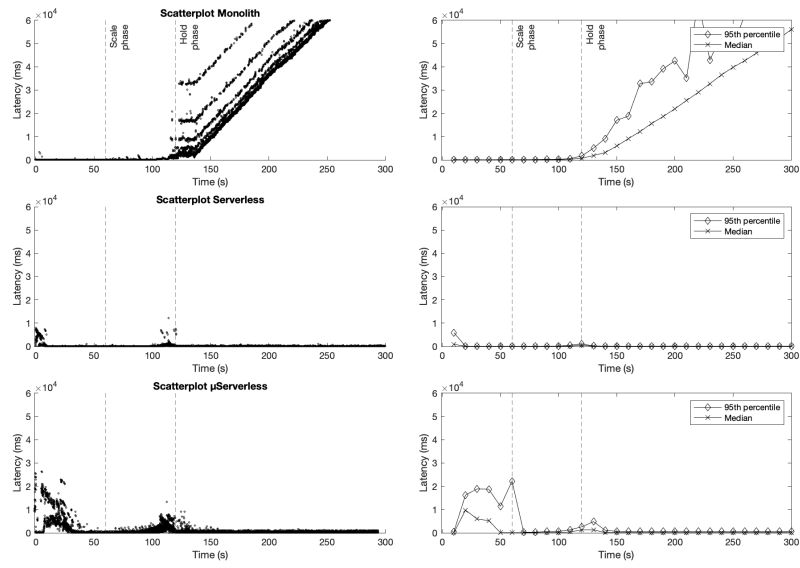


Figure 30 Scenario 1 Workload 3

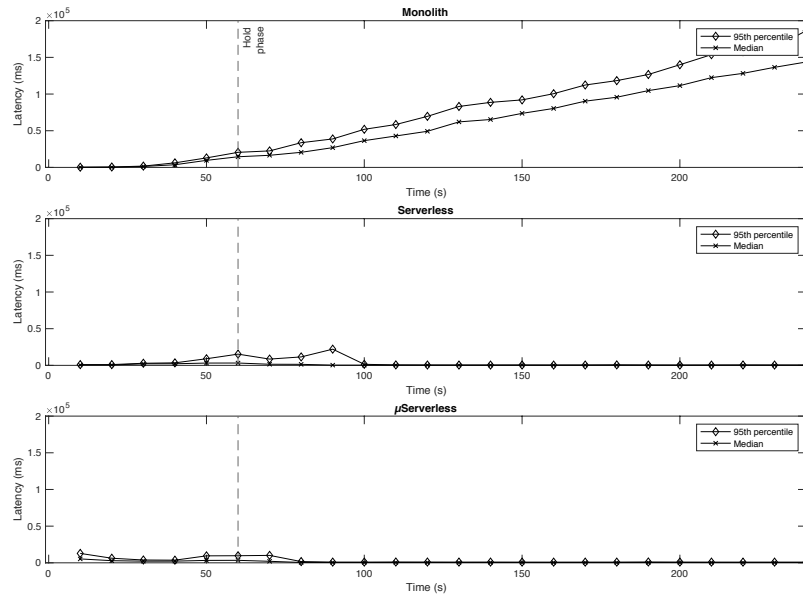


Figure 31 Scenario 2 Workload 1

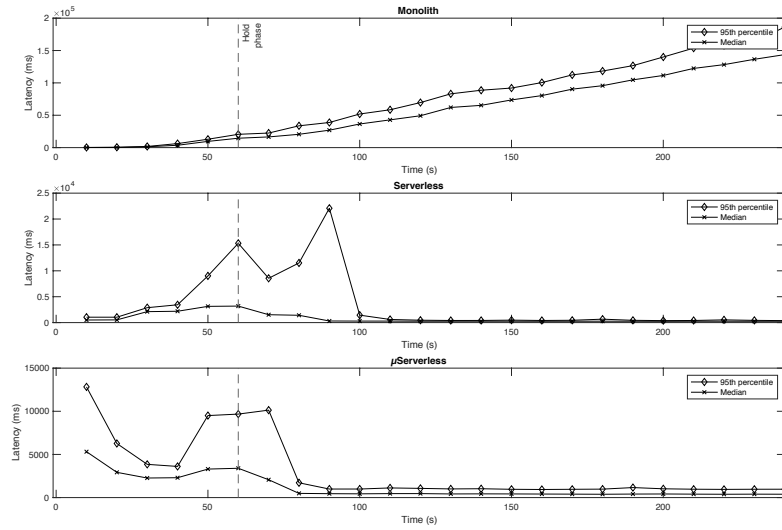


Figure 32 Scenario 2 Workload 2

A.3 Definition of identified variables

Table 15 Identified Variables

Variable	Summary
FaaS Cloud Provider (Azure)	The cloud provider offering FaaS services.
Cold start	The increase in response time from allocating resources and spinning up a function instance.
Package Deployment	Deployment of a serverless application as a packaged file.
Non-Package Deployment	Deployment of a serverless application as source code.
Reliability	The percentage of successful request
Scalability	The ability to start new instances in response to increased load.
Extensibility	The ability to extend a system in terms of effort and effect on the system structure.
Architecture	The architectural design of a system.
Cost	The billing model for serverless and non-serverless hosting.

Infrastructure management	The effort of setting up and managing infrastructure such as servers.
Vendor lock in	The degree to which an implementation is tied to the cloud provider
Granularity	The degree to which a system is split into independent services.