Soeun Jang

The Feasibility of Function as a Service to Improve Backend

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

30 November 2018

Author Title Number of Pages Date	Soeun Jang The Feasibility of Function as a Service to Improve Backend 35 pages 30 November 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, The head of department Sumanta Saha, The chief engineering architect

Since cloud computing emerged, business paradigm was changed more dynamically, thus application architecture was required to have a capability to be interact with new requirements as fast as possible. Therefore, Microservices has been the trend of application architecture due to its flexibility. However, some of issues have been surfaced and the movement to solve those issues have been processing. Recently, cloud computing providers such as Amazon, Microsoft, IBM, and Google introduced the possible solution that is Function as a Service.

The paper aimed to examine whether FaaS can be improved application backend via integrating FaaS architecture into Microservices architecture. For implementation, two prototypes were designed. One is the prototype of Microservices backend was implemented. The other is Microservices combined with FaaS that some of services in the prototype of Microservices were changed to FaaS via Kubeless framework, one of Serverless framework. The result of benchmarking both prototypes was that Microservices performed better than Microservices combined with FaaS in terms of backend performance. On the other hands, CPU and memory usage and development cycle was more efficient in Microservice combined with FaaS than Microservices.

To summarize, FaaS did not improve backend performance but improve CPU and memory usage and development cycle. Therefore, readers need to consider which point is important in their use case to decide incorporating FaaS into their system.

Keywords	Function as a Service, Microservices, Backend, Cloud computing, Software architecture

Contents

List of Abbreviations

API	Application Programming Interface.
AWS	Amazon Web Service.
CD	Continuous Deployment.
CI	Continuous Integration.
FaaS	Function as a Service.
IaaS	Infrastructure as a Service.
PaaS	Platform as a Service.
SaaS	Software as a Service.

1 Introduction

Cloud Computing emergence changes business paradigm for companies and organizations, allowing to avoid issues about maintainability, scalability, cost, and so on [1, p1]. IT companies jump into cloud computing technology and various companies incorporate cloud computing into their business via cloud computing providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform.

After cloud computing emerged, application architecture transformed from Monolithic architecture to Microservices. Microservices architecture solves traditional problems in terms of integration, deployment, scaling, and so on. While single instance contain multiple business contexts in Monolithic architecture, Microservices architecture partitions Monolithic single instance into multiple independent instances called service having one business context [2, p89].

Despite of Microservices' efficiency, some of issues are present in Microservices architecture. For instance, issues are high complexity of system, redundant resource consumption, duplicated codes, and other challenges in terms of monitoring and debugging and end-to-end testing [2, 3]. Meanwhile, Function as a Service (FaaS), known as Serverless computing, has been trend of application architecture due to the possibility of solving Microservices issues. Moreover, companies have started utilizing FaaS in their infrastructure because of advantages such as serverless concept, cost-saving, and fast development cycle, and so on [4, 5].

Therefore, this paper aims to examine whether FaaS can be improved application backend via incorporating FaaS into Microservices. For evaluating the result of the incorporation, CPU and memory usage, backend performance testing, and examination of development cycle were carried out. The topic of the paper was motivated by LMF Oy Ericsson Ab, one of the providers of Information and Communication Technology (ICT) to service providers, with mobile traffic carried through its networks. The reference sources in this paper were collected from reliable sources such as Safari Books Online, Google Scholar, IEEE, and other research papers.

2 Literature Review

Chapter 2 Literature Review aims to explore Microservices and FaaS architecture to establish the background for understanding the proposed solution in further chapter. This paper starts with cloud computing. Cloud computing established the basis for Microservices and FaaS. Next to cloud computing, general information of Microservices and FaaS are provided to understand how each architecture works. Furthermore, benefits and drawbacks of Microservices and FaaS are presented, which gives more insight of how architecture style can improve backend.

2.1 Cloud Computing

Cloud computing is “applications and services that run on a distributed network using virtualized resources and accessed by common Internet protocols and networking standards” [6, p3]. Before cloud computing, company needed to buy hardware and other machineries required for system. Moreover, company had responsibilities of system maintenance and scalability. Since cloud computing emerged, cloud technology has rapidly seeped into business, reducing efforts from company and changing the procedure of building and deploying system from traditional way to cloud way [7, p1]. In addition, abstracted and virtualized system of cloud computing are concealed from users such as developers and other clients and eliminates the user’s responsibilities regarding physical system and resource management [6, p4].

Companies are encouraged to incorporate cloud computing service into their business. Due to five main characteristics of cloud computing, the entry level of business got lower and cheaper so that not only big company, but also small and medium company can provision Information Technology (IT) into their business [6, p18]. First main characteristic of cloud computing is that a user can utilize services on demand such as provisioning resources, and so on [6, p17]. Second main characteristic is high accessible network, so that a user can access cloud through network [6, p17]. Third and fourth main characteristics are regarding resource [6, p17]. Resources of cloud computing are dynamically allocated and reallocated on demand and highly elastic to be provisioned. Finally, resources used by a user can be visualized via metering system [6, p17]. Other benefits of cloud computing are low costs, easy-to-use, high reliability, scalability, and maintainability [6, p17, p18].

Furthermore, cloud computing not only has benefits for existing business model but also provides different types of deployment models and service models.

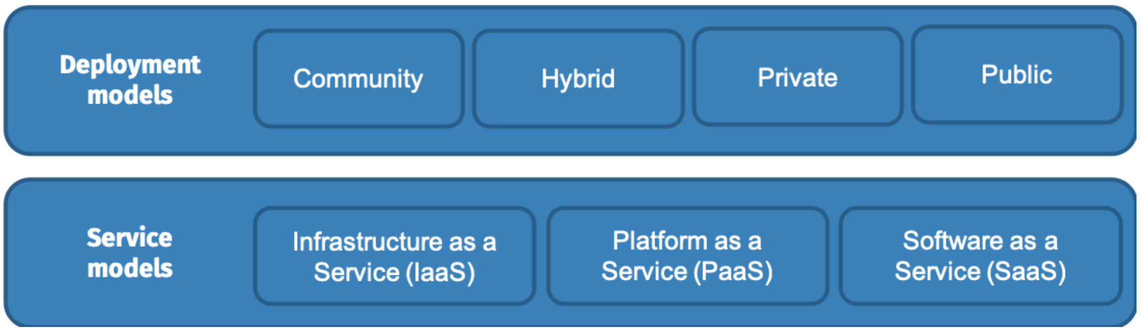

Deployment models
Community
Hybrid
Private
Public
Service models
Infrastructure as a Service (IaaS)
Platform as a Service (PaaS)
Software as a Service (SaaS)

Figure 1. Deployment and service models of The NIST Cloud Computing definitions [6, p6]

Above Figure 1 illustrates deployment models and service models among the NIST cloud computing definitions. The types of deployment model are public cloud, private cloud, hybrid cloud, and community cloud. The types of service model are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Deployment models relate to define the location and management of infrastructure of the cloud [6, p5]. On the other hands, service models relate to use cloud computing platform [6, p5]. Different abstraction of each cloud service is provided and reduces required resources to develop systems. Therefore, companies can determine proper service model based on their business type.

The main subject of this paper is Function as a Service (FaaS) which is close to service models. Therefore, this chapter focus on service models of cloud computing. Below Figure 2 illustrates cloud service models.

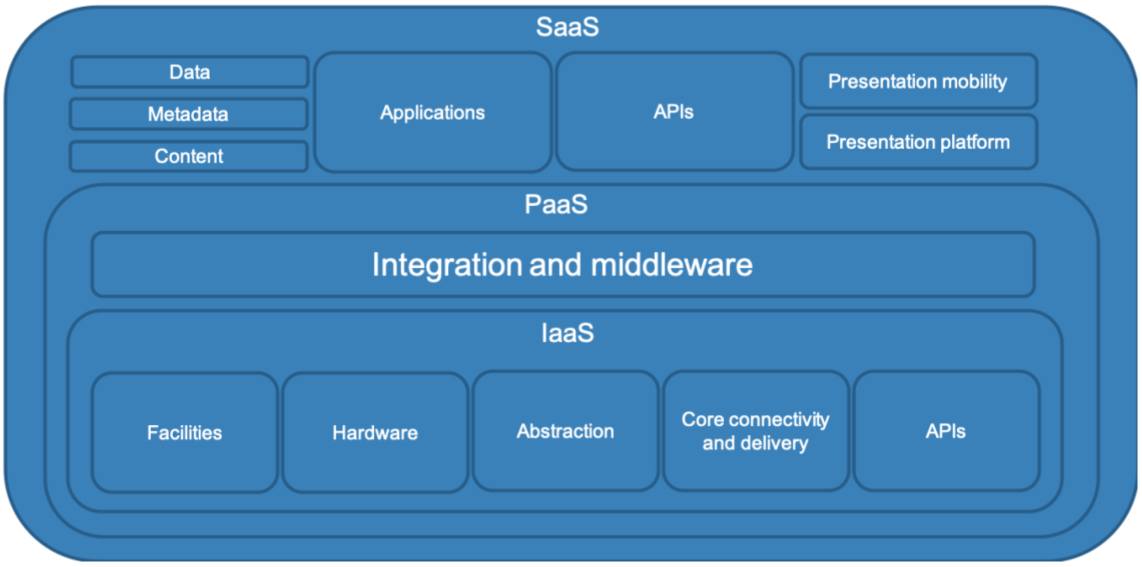


Figure 2. The cloud service models [6, 11p]

Above Figure 2 describes detail resources and infrastructure of each cloud service model. Among service models, IaaS provides infrastructure to users, consisting of facilities, hardware, abstraction, core connectivity and delivery, and APIs. IaaS service model can eliminate concerns about management of infrastructure and users are only demanded to cope parts of all deployment [6, p10]. On the other hands, users of PaaS service model are provided platform which is added integration and middleware to IaaS. Therefore, users can use the infrastructure or application provided by PaaS and are responsible of installation and management of the application [6, p10]. In contrast, SaaS service model provides a complete application to users, adding data, metadata, content, applications, APIs, presentation mobility, and presentation platform to PaaS. The application is managed by SaaS service providers and users take responsibility of data handling and user interaction [6, p10].

Since cloud computing has utilized in general, a movement of transformation from Monolithic architecture to Microservices architecture happened. In addition, cloud computing boosted fully-implemented Microservices architecture. Next chapter focuses on Microservices architecture.

2.2 Microservices

Before cloud computing, Monolithic architecture style had been the standard architecture to build system. Since cloud computing traded the paradigm of business and has given effects on the way of development, Microservices is one of popular and well-known in application architecture field [1, p1]. Even though applying Monolithic architecture on top of cloud is simple, the movement of transformation from Monolithic to Microservices rapidly surfaced.

2.2.1 Overview of Microservices

Monolithic architecture is simple and single instance which contains all business contexts. Single instance of Monolithic architecture contains UI layer, application logic layer, and database layer [8, p5]. Therefore, developing a small application is easier and cheaper than using other concepts or interfaces [8, p5-7]. However, if the application size is increased, continuous integration and deployment are demanding due to increased application complexity [8, p8]. Therefore, slow development cycle, high deployment complexity, and low scaling capability are issued [2, p2]. Solving those issues, Microservices architecture style has derived new paradigm to reduce development cost and time.

Microservices breaks down single instance to multiple independent instances called service [2, p89]. Each service is autonomous, bound by one business context without knowing other services' implementation [2, p3]. Thus, each service is independently deployable and scalable [2, p10]. Below Figure 3 demonstrates abstracted Microservices architecture to understand more detailed Microservices architecture.

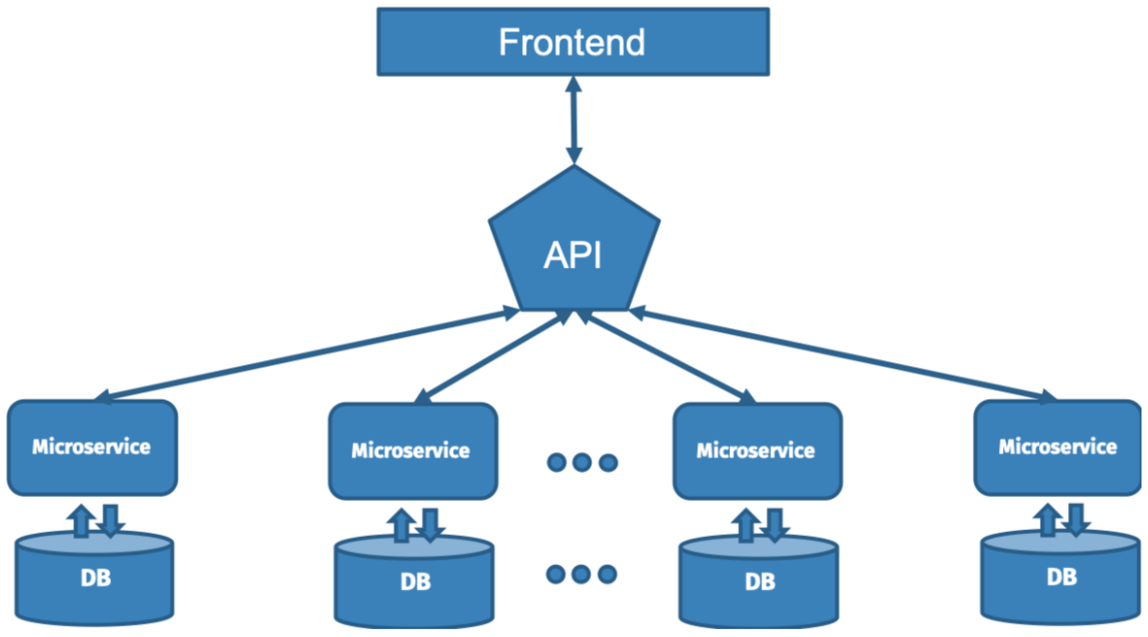


Figure 3. Example of Microservices architecture

Above Figure 3 is example of Microservices application architecture. Figure 4 can be divided three parts such as database, communication between Microservices, and communication between backend and frontend. First, each Microservice own a database. Each Microservice database works independently without impact to other Microservice database. Second, each Microservice communicates through Application Programing Interface (API) such as http request. This communication way achieves independent presence of each Microservice. Third, communication between backend and frontend operates through API call. The benefits of Microservices from the architecture are presented in the following chapter 2.2.2.

2.2.2 Benefits and Drawbacks

Benefits

Among multiple benefits of Microservices, three key benefits are mentioned in this chapter. Three key benefits are in terms of compact and independent service, integration and deployment, and scaling [2, p4-8].

First key benefit is compact and independent service [2, p4]. Compact and independent service can improve in multiple aspects of development. While Monolithic application is normalized, Microservices guarantees that developers can select proper technologies for each service because each Microservice is independent, separating with others. In addition, integrating new service or feature into Microservice application is highly available and more quickly than into Monolithic application, thus Microservices has higher capability to handle changes. [2, p4.] Furthermore, the benefit leverages the resilience of application. Since each service can work individually without recognizing the presence of other services, if a service is failed or crashed, the rest of service work without any impact from the service. [2, p5.]

Next key benefit is about integration and deployment in Microservices. During whole development process, developers occasionally integrate codes, work on different features, modify a service, and release new feature at the same time. This means developers are integrating and deploying continuously in every working day. Glancing Continuous Integration (CI) and Continuous Deployment (CD) in Monolithic application seems simple, but even small change such as changing one-line code is required to deploy entire application to apply the change. In contrast, only relevant service with the change is integrated and developed in Microservices architecture. [2, p6.] For instance, team A is working on service A. When changing or adding new feature to service A, team A needs to build and test changes in service A's CI build. The whole process does not give any impact to irrelevant services and require integrating and deploying the rest of services. Therefore, the way of Microservices integration and deployment increases speed of delivery with new features and reduces time of development cycle.

Finally, last key benefit of Microservices is in terms of scaling. Each Microservice can be scaled separately, thus scaling is horizontal, less hardware power, less cost, and less time to scale [9, p2-3]. On the other hand, Monolithic application scales everything at the same time [2, p5].

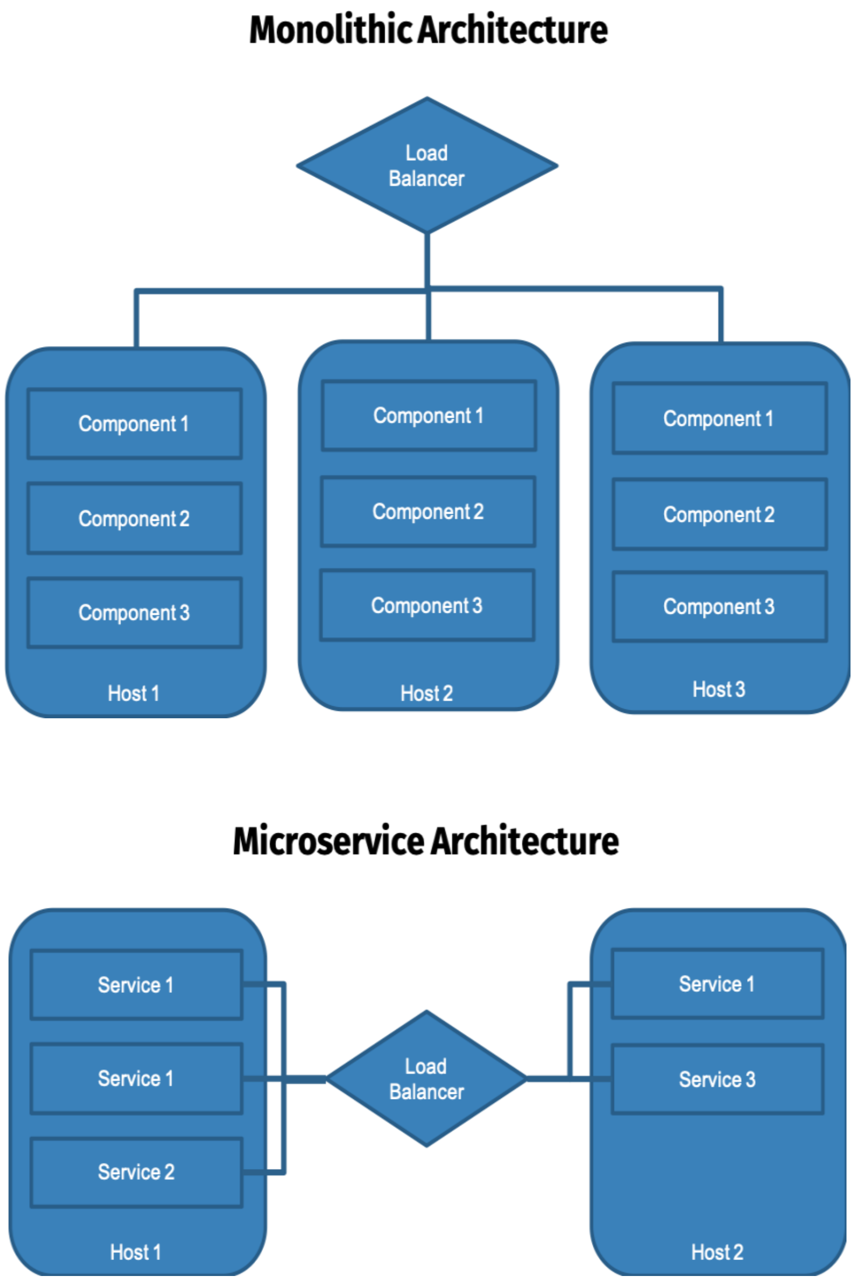


Figure 4. Monolithic architecture and Microservices architecture scaling [9, p4]

Above Figure 4, top figure describes Monolithic scaling example. In Monolithic architecture scaling, single load balancer is mapped to all units to take responsibility of scaling. Due to single instance, Monolithic load balancer does not know which service of unit is needed for which context so that Monolithic load balancer duplicates all units instead of creating required units. In contrast, Microservices architecture application has load balancer mapped to certain units. [9, p3-4.] Above Figure 4, bottom figure describes

Microservices scaling example. Due to independent service with one context, load balancer identifies each service's context. Therefore, Microservices load balancer scales certain service horizontally [9, p3-4].

Drawbacks

Even though Microservices architecture improves diverse aspects of application architecture, some of issues are remained. First issue is Microservices increases complexity of system [3, p131]. Maintaining each service individually adds latency of complex development and operation. Second is redundant resource consumption in case of dynamic workloads. Even though some of services are not used often, all services are running constantly on the cloud, causing extra resource consumption and costs. Third is duplicated codes in Microservices [2, p59]. To achieve loose coupling Microservices prefers duplicate codes in each service rather than shared functions [2, p23]. However, duplicated codes enlarge the size of application unnecessarily. Fourth issue is difficulty in monitoring and debugging. Due to distributed Microservices architecture, developers monitor and debug each independent Microservice, making tracing the cause difficult [10, p4]. Moreover, Microservices has low capability of end-to-end testing [2, p138]. Below Figure 5 demonstrates end-to-end test in Microservices.

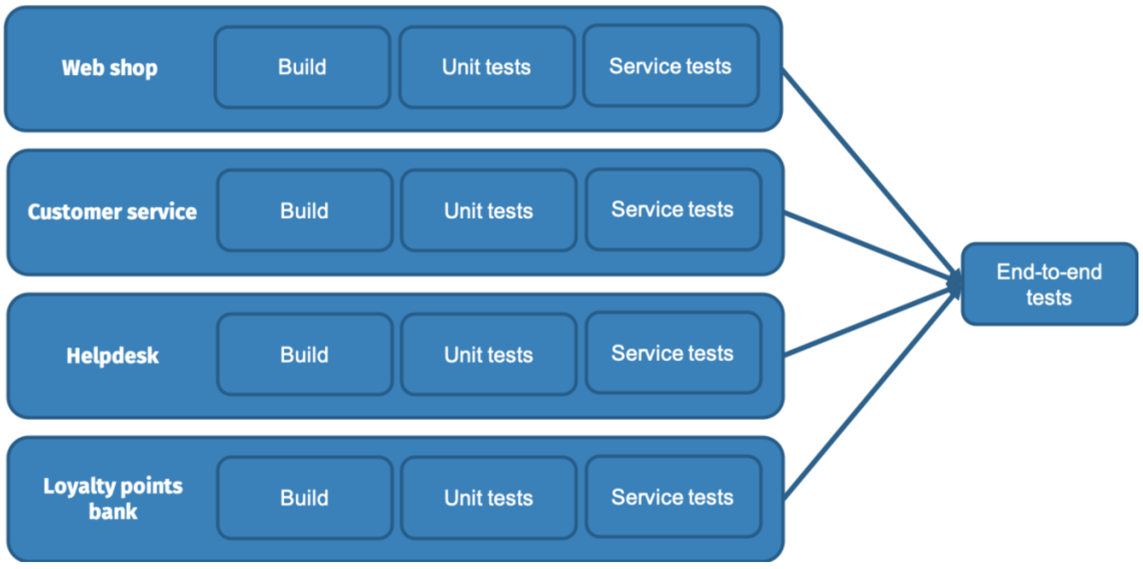


Figure 5. Example of end-to-end test in Microservices [2, p139]

Above Figure 5 illustrates an example of end-to-end test in Microservices. For end-to-end tests in Microservices, deployed multiple services are tested at the same time [2, p139]. Each service builds, runs unit tests and service tests separately. Then, end-to-end tests run against all services. Whenever a service is changed, developers need to deploy all services and run end-to-end tests, causing further development cycle such as integration and deployment delayed [2, p139].

For these reasons, the movement of finding new concepts has been arisen to resolve issues from Microservices. The possible solution is FaaS. Currently, Amazon Web Services, Google, and Microsoft introduced FaaS as Amazon Web Services Lambda, Google Functions, and Microsoft Azure Functions. Therefore, next chapter focuses on FaaS in general to perceive how FaaS could be the solution to improve Microservices.

2.3 Function as a Service

While Microservices has been mature and well-known architecture in software industry, some improvements present in Microservices architecture. Meanwhile, introducing by cloud computing providers, FaaS is considered as possible solution to enhance Microservices application.

2.3.1 Overview of Function as a Service

Since business is more complex and dynamic, the application architecture flow has been changed to split one application logic into smaller pieces. FaaS introduced smaller unit than Microservices' unit, maximizing the capability of handling new requirements. Below Figure 6 demonstrates transformations of application architecture.

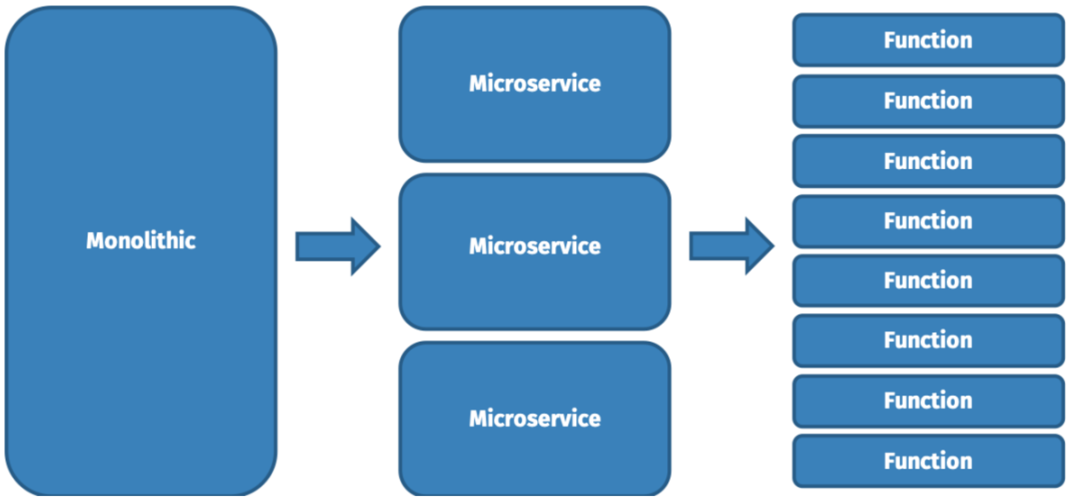


Figure 6. Monolithic, Microservices, and FaaS architecture [5, p9]

Above Figure 6 is abstraction of each architecture such as Monolithic architecture, Microservices architecture, and FaaS architecture. From traditional to modern architecture, the size of unit is smaller because segmentation makes application reflect requirements as quickly as possible. Monolithic architecture has single application logic contains all business contexts [8, p5]. Microservices architecture divides one application logic into multiple services which contain single business context [2, p89]. Then, FaaS architecture splits a service into multiple functions [11, Chp1]. Segmented FaaS functions can be deployed, invoked, and run with fully management and maintenance by cloud computing provider. Therefore, developers can escape from the obligation of provisioning or managing their infrastructure. In addition, companies or clients can focus on developing their business. [12, p8.]

Moreover, FaaS functions pursue event-driven paradigm [13, p8]. Event-driven paradigm can prevent consuming redundant resource and expense and achieve loose coupling. Each function is invoked when the event is triggered. [5, p10.] For instance, each endpoint can be isolated as function and each function work highly on-demand, not running continuously [14, p9]. Due to event-driven characteristic, FaaS is suitable for use cases such as application backend, high transactions such as data processing and image processing, and scheduled jobs [11, Chp1]. Below Figure 7 demonstrates how FaaS functions work in the architecture point of view.

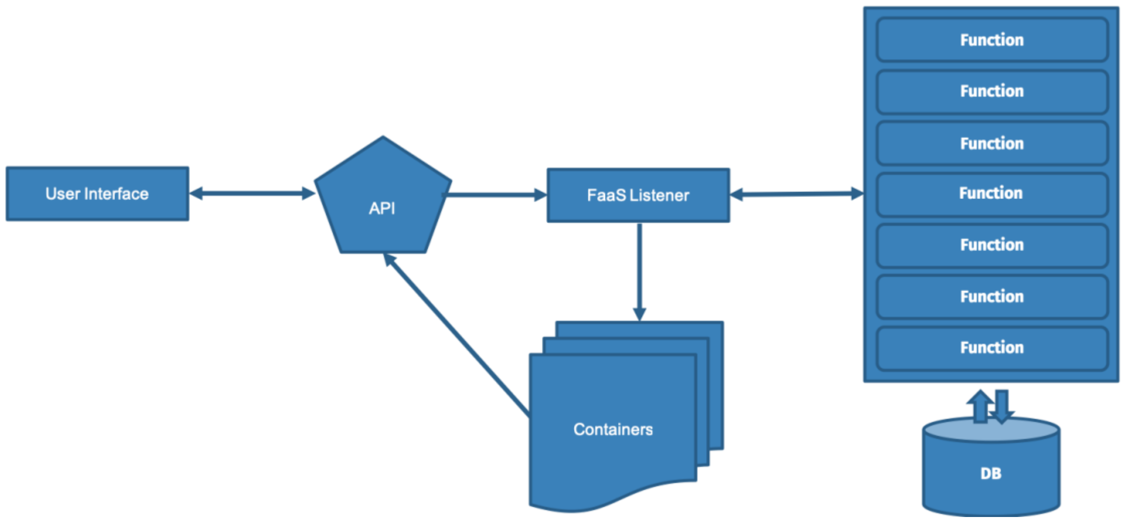


Figure 7. FaaS application architecture

Above Figure 7 illustrates an example of FaaS application architecture. FaaS architecture consists of FaaS event listener, list of functions, and containers to run functions. While FaaS event listener is waiting for events, nothing is running on Kubernetes cluster. When the event listener catches event, the listener dispatches correct functions from the list of functions and runs on each function’s runtime container on Kubernetes cluster. After the function completes job, functions are removed from the container. Compared to Microservices’ mechanism that each Microservice runs continuously on Kubernetes cluster, this mechanism can handle request in economical and effective way.

2.3.2 Benefits and Drawbacks

Benefits

Due to several benefits of FaaS, the demands to integrate FaaS has increased. Among multiple benefits of FaaS, three main benefits are introduced in terms of no server, cost, and development.

First main benefit is no server in FaaS. No server does not mean that physical server is not present but no server to be managed by developers. While business barrier has been lowered after cloud computing, developers are still responsible to maintain systems such as installation, configuration, management, and so on [4, p8-9]. Since FaaS has emerged, developers simply deploy on the cloud environment which is set up by cloud computing provider. Therefore, needs to know and maintain the system can be eliminated from developer’s daily work by FaaS [4, p8-9]. In addition, the first benefit of FaaS can save time and cost to maintain servers [11, Chp1].

Second benefit of FaaS is in terms of cost from three parts such as operation, scale, and computing. First, FaaS can reduce operational cost due to cloud provider’s maintenance. Clients save expenses caused by buying machineries and managing the infrastructure. [5, p17.] In addition, FaaS can reduce cost regarding scaling due to the ability of auto-scaling in FaaS [5, p18]. Cloud provider is responsible to take care of scaling so that developers can keep from complex scaling procedures. Moreover, cost of computing resources can be lessened in FaaS [13, p13]. By using FaaS architecture, pay-per-use billing model is fully available. Each function pursues event-driven paradigm and the paradigm achieves pay-per-use based on consumption and execution, not resources. [4, p9-10.] Therefore, enterprise or developer only pays for what they consume excluding idle resources [12, p9].

Finally, main benefit in FaaS is rapid development. Building FaaS application means endpoint developers do not know and need to worry about the infrastructure [4, p9]. FaaS handles functions for building an application on the cloud platform in contrast to services in Microservices. Endpoint developer is simply deploying and invoking new functions, mapped with certain events [14, p9-10]. The simple process of development achieves template development so that development can be easier and faster. Moreover, Command-Line Interface (CLI) or Graphic-User Interface (GUI) serverless frameworks such as Kubeless, OpenFaaS, and Fission support makes on-boarding easily. In addition, FaaS architecture supports multiple runtimes to develop polyglot functions so that developer can choose different runtime for each function to make codes more efficient [12, p9].

Drawbacks

Despite of the advantages, challenges exist in FaaS. Among challenges in FaaS, four critical challenges are discussed in this chapter. First challenge is high latency caused by cold start. FaaS functions are event-driven which means the functions are not running constantly. Therefore, when a function is invoked first time, from preparation to invocation takes multiple steps such as finding the function, dispatching function into a container, and running on the container. Those steps add high latency in FaaS architecture [13, p17].

Second challenge of FaaS is low capability of debugging and monitoring. Unit testing and integration testing in FaaS are simpler than Microservices [13, p19]. However, debugging and monitoring is more difficult because of stateless and independent functions [13, p20]. Therefore, developers need to examine logs of functions which makes difficult to debug and monitor an issue or bug [13, p19-20].

Third challenge is limited execution time of function [5, p13]. FaaS functions were designed to be efficient when the execution time is short [13, p18]. Therefore, using FaaS is efficient in terms of cost in case of backend, high transactions job, and scheduled jobs [11, Chp1]. If the characteristic of job is long duration, the cost is high so that using FaaS can be meaningless for that case [13, p18].

Finally, FaaS contains vendor dependency [13, p19]. For incorporating FaaS into existing solution, companies need to use third-party, such as Amazon Web Services, Microsoft Azure, or other frameworks. Each vendor contains different rules, regulations, billing system, and so on, causing difficulty of changing another vendor [13, p18]. In addition, if a vendor has problems, companies which use the vendor are influenced critically to their business [13, p19].

The goal of the literature review is to explore Microservices architecture and FaaS to establish knowledge for understanding the proposed solution in further chapter. Firstly, Microservices has fairly enough small instances than Monolithic’s single instance, but some of instances could be isolated and scaled separately, which could be resolved by FaaS. Secondly, FaaS is beneficial for application backend due to highly event-driven characteristic. In addition, FaaS achieves pay-per-use model, solving redundant resource consumption. Therefore, each architecture’s weaknesses can be enhanced by each other. Building Microservices combined with FaaS could be the proposed solution in the paper. Chapter 3 Methods and Materials explains detailed proposed solution.

3 Methods and Materials

The goal of chapter 3 Methods and Materials is to dedicate implementation of Microservices combined with FaaS. In this chapter, entire project is building the prototype of Microservices backend and Microservices combined with FaaS backend. This chapter explains overall and detailed information about solution. Following Figure 3 describes how solution was implemented.

Solution Workflow

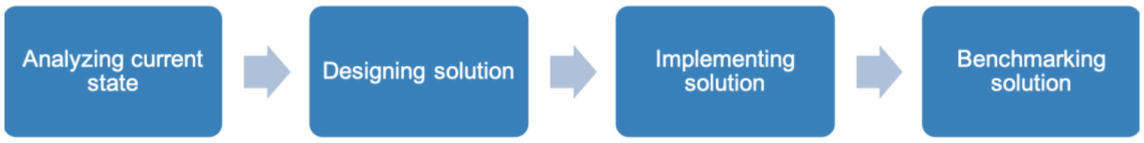


Figure 8. Solution workflow

Above Figure 8 illustrates solution workflow. Solution workflow processed in four steps such as analyzing current state, designing solution, implementing solution, and benchmarking solution. First, analyzing current state clarified weaknesses of current solution to understand which points could be improved via proposed solution. Next, designing solution was deciding overall and detailed factors of proposed solution regarding architecture of new solution, frameworks, tools, and so on. Third step was implementing proposed solution based on design step. Finally, benchmarking solution showed the results to determine feasibility of incorporating FaaS into backend architecture in chapter 4 Results.

3.1 Current State Analysis

Prototype of current solution is based on Microservices architecture and has four Microservices in Kubernetes cluster. Technical stacks for prototype of current solution are Golang, MySQL, Kubernetes, and Docker. Microservices backend is written in Golang with MySQL database and deployed in Kubernetes cluster. Docker is used for pushing and pulling images of Microservices.

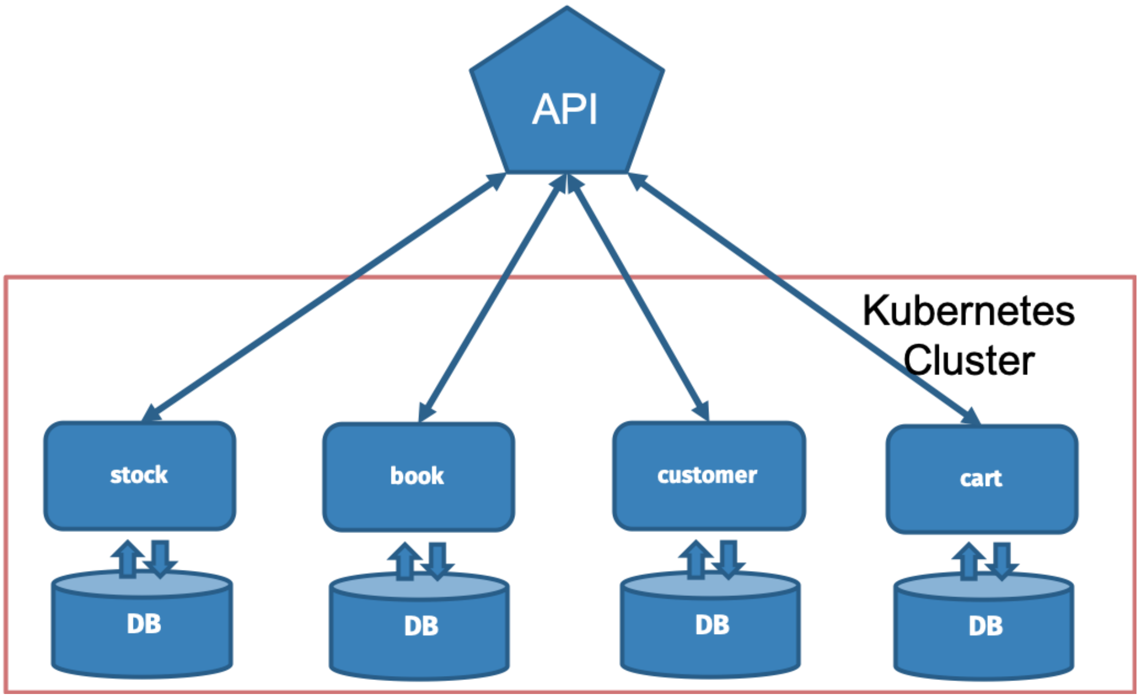


Figure 9. Microservices backend architecture

Above Figure 9 is Microservices backend architecture for prototype of current solution. For achieving independent presence of Microservices without knowing each other, four Microservices such as stock, book, customer, and cart are detachedly deployed in the Kubernetes cluster and own separate database. In addition, a Microservice is a pod running constantly in Kubernetes cluster. Each Microservice communicates with frontend and other Microservices through API gateway. Example of Microservices backend endpoints are in the following list.

- GET http://192.168.99.100:30067/books : to read all books
- GET http://192.168.99.100:30067/book/:id : to read book by id
- POST http://192.168.99.100:30067/book : to post new book
- UPDATE http://192.168.99.100:30067/book/:id : to update book
- DELETE http://192.168.99.100:30067/book/:id : to delete book
- GET http://192.168.99.100:30068/stocks : to read all stocks
- GET http://192.168.99.100:30068/stock/:id : to read stock by id
- POST http://192.168.99.100:30068/stock : to post new stock
- UPDATE http://192.168.99.100:30068/stock/:id : to update stock

Through those endpoints, CRUD is operated and a Microservice can obtain other Microservice information. For example, Microservice cart needs customer name and book title for creating new cart. Microservice cart sends requests to Microservice customer and book to derive customer name and book title. Below Listing 1 explains example through source code.

```
func PostCart(c *gin.Context) {
	var cart Cart
	var book Book
	var customer Customer
	c.Bind(&cart)
	cid := c.Params.ByName("customerid")
	bid := c.Params.ByName("bookid")

	geturl := fmt.Sprintf("http://192.168.99.100:30066/books/%s/title", bid)
	response, err := http.Get(geturl)
	if err != nil {
		//error handling
	}
	defer response.Body.Close()
	bodyBytes, err := ioutil.ReadAll(response.Body)
	err = json.Unmarshal(bodyBytes, &book)
	if err != nil {
		//error handling
	}
	geturl = fmt.Sprintf("http://192.168.99.100:30065/customers/%s/name", 
cid)
	response, err = http.Get(geturl)
	if err != nil {
		//error handling
	}

	defer response.Body.Close()

	bodyBytes, err := ioutil.ReadAll(response.Body)
	err = json.Unmarshal(bodyBytes, &customer)
	if err != nil {
		//error handling
	}
	stmt, err := db.Prepare("insert into carts (customername, booktitle) 
values(?,?);")
	if err != nil {
		//error handling
	}
	_, err = stmt.Exec(customer.Result.Name, book.Result.Title)
	if err != nil {
		//error handling
	}

	c.JSON(http.StatusOK, gin.H{
		"message": "successfully created",
	})
}
```

Listing 1. Example code of Microservice handler function PostCart

Above Listing 1 displays example source code of Microservices handler function PostCart. Function PostCart operates POST method and reacts to http request, POST http://192.168.99.100:30066/cart/:customerid/:bookid. PostCart function has two important parts to understand communication with other Microservices. Posting new cart requires customer name and book title. Therefore, fist part is sending http request to Microservice book to derive matching book data. Second part is sending http request to Microservice customer to derive matching customer data. Using collected data, cart creates new cart to database carts. Through this process, Microservices can obtain and use other Microservices data.

This current state has three main improvements. First, all Microservices need to work independently, so that each Microservice has own database and excessive duplicate codes. This enlarges the size of Microservice than ideal small. Secondly, development cycle can be improved. For adding new Microservice, endpoint developers write codes for new Microservice, integrate them, handle docker file and image for new Microservice, and deploy and test new Microservice in Kubernetes. This process of development slow down with extra time consumption. Third, some of Microservices corresponds to dynamic workload. This means that some of Microservices are not necessary to run constantly in the cloud with extra expenses. This paper assumes that this current solution contains two Microservices which are dynamic and burstable workload. Therefore, next chapter 3.2 proposes and implements new solution for improving three points.

3.2 Proposed Solution

Proposed solution is based on Microservices architecture combined with FaaS architecture to improve some of each other drawbacks. In addition, technical stacks for the proposed solution are Golang, MySQL, Kubernetes, and Docker as well as current solution. In addition, FaaS framework is used to building FaaS architecture for the proposed solution.

FaaS framework

Amazon web service (AWS), Google Cloud, and Microsoft Azure are familiar providers of FaaS enterprise framework. However, few researches of FaaS exists with open source framework which is contributed by individuals. Therefore, solution used one of open source framework to build FaaS. Below Table 1 illustrates FaaS frameworks such as Kubeless and OpenFaaS.

FaaS framework	**Kubeless**	**OpenFaaS**
Kubernetes-native framework	Yes	Yes. Docker and Kubernetes
Supported runtimes	Python, NodeJS, Ruby, PHP, Golang, .NET, custom runtimes	Not limited. Any Docker container
Serverless compatibility	Yes	Work in progress
License	Apache-2.0	MIT

Table 1. Comparison of FaaS frameworks

Above Table 1 compares FaaS frameworks. Kubeless framework is a serverless framework licensed by Apache-2.0 [15]. In addition, Kubeless framework supports not only runtimes such as Python, NodeJS, Golang, and .NET but also custom runtime [15]. Kubeless framework is built based on native Kubernetes, leveraging function management on Kubernetes environment [15]. Especially, Kubernetes' Customer Resource Definitions can be deployed and exported by Kubeless framework [15]. In addition, Kubeless framework provides event trigger, serverless plugin, auto-scaling, and monitoring features [15]. Therefore, Kubeless framework has low barrier to existing users who use Kubernetes which means optimizing learning curve and changes of existing system.

On the other hand, OpenFaaS framework is a serverless framework with Docker and Kubernetes licensed by MIT [16]. OpenFaaS framework supports any Docker container, however serverless plugin is still in progress [16]. In addition, UI portal provided by OpenFaaS framework lets users to manage functions such as creation, invocation, deletion, and configuration and provides packaging feature to consume any process as function [16]. Moreover, functions can be written in any language using any Docker containers. Furthermore, templated function management is possible by CLI with YAML and auto-scaling on-demand is possible in OpenFaaS framework [16].

In spites of OpenFaaS framework's wide capability, Kubeless framework is chosen as a serverless framework in this paper because Kubeless framework is friendlier with native Kubernetes and uses only YAML file so that the usage of the framework is simpler and easier than OpenFaaS framework. Therefore, proposed solution uses Kubeless framework to implement FaaS in backend. Using Kubeless framework, below Figure 7 displays detailed solution architecture.

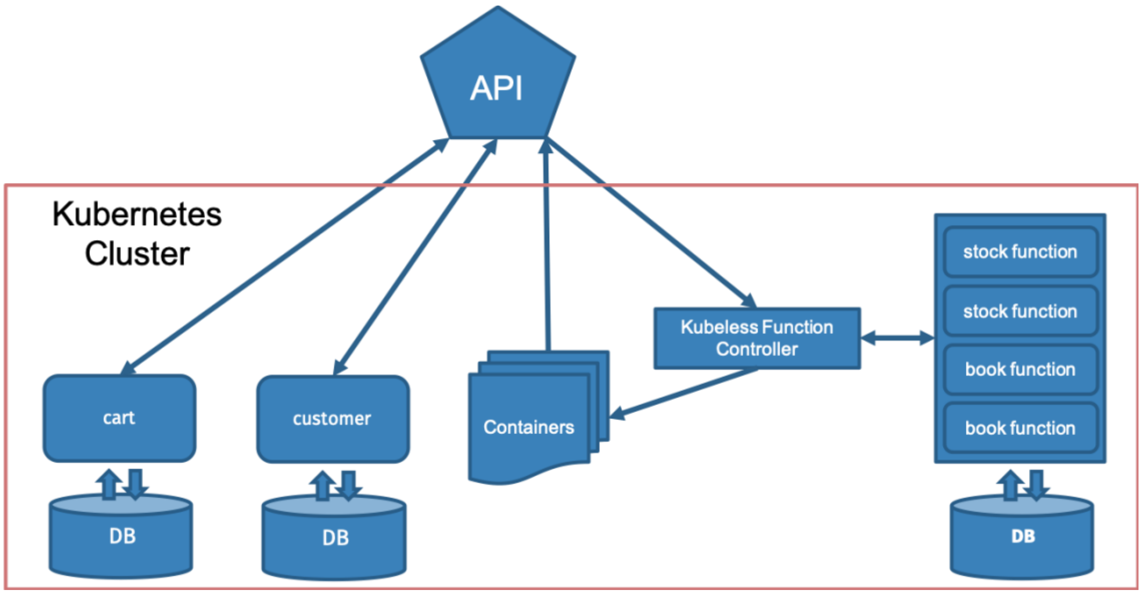


Figure 10. Proposed solution architecture

Above Figure 10 illustrates the proposed solution architecture. The solution architecture consists of two Microservices and FaaS functions which are split from Microservices stock and book, assuming that Microservice stock and book are dynamic workload jobs. FaaS architecture comprises Kubeless Function Controller, functions, database, and containers. When function is called through HTTP request, Kubeless Function Controller catches the request and invokes proper function in a container. Examples of changed endpoints are in the following list.

- http://192.168.99.100
  --header 'Host': 'getbooks.192.168.99.100.nip.io': to read all books
- http://192.168.99.100?id
  --header 'Host': 'getbookbyid.192.168.99.100.nip.io': to read book by id
- http://192.168.99.100
  --header 'Host': 'postbook.192.168.99.100.nip.io': to post new book
- http://192.168.99.100?id
  --header 'Host': 'updatebook.192.168.99.100.nip.io': to update book
- http://192.168.99.100?id
  --header 'Host': 'deletebook.192.168.99.100.nip.io': to delete book

In contrast to Microservices endpoints, FaaS endpoints use host for Kubeless Function Controller to distinguish each function and invoke corresponding function, so that functions can be called without specific methods such as GET, POST, UPDATE, and DELETE. In file, small and stateless FaaS functions are listed without router or built-in libraries to manage server. Listed functions are assigned by Kubeless framework when the function is deployed in Kubernetes cluster. In addition, for enabling http trigger, each function needs to add http trigger via ingress and obtain new host. Using the host, function can be called by http request. Moreover, Kubeless framework provides a build process for functions. Below diagram is about the build process in Kubeless.

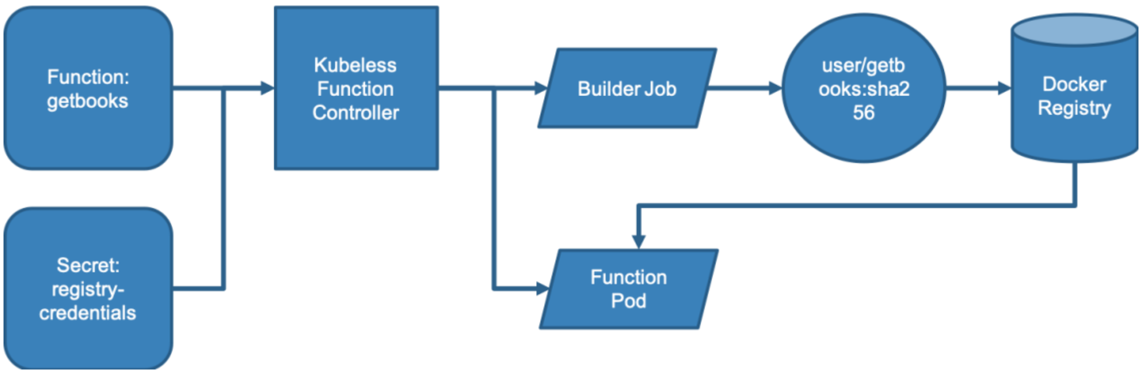


Figure 11. Kubeless framework's build process for functions [17]

Above Figure 11 illustrates Kubeless build process to increase speed of redeployment and provide immutable function deployment [17]. First step is creating a docker image for a function and storing the docker image in a docker registry. Then, Kubeless Function

Controller generates builder job and function pod. Builder job is pushing the docker image to repository with registry credentials. Function pod is pulling the docker image for function and running the function. Therefore, when builder job is completed, function pods job can be started. Through this process, a function can be deployed in Kubernetes. Following List 2 is actual example code of a function in FaaS.

```
func PostStock(event functions.Event, context functions.Context) (string,
error) {
    var stock Stock
    err := json.Unmarshal([]byte(event.Data), &stock)

    stmt, err := db.Prepare("insert into stock (isbn, count) values(?,?);")
    if err != nil {
        fmt.Print(err.Error())
    }
    _, err = stmt.Exec(stock.ISBN, stock.Count)
    if err != nil {
        fmt.Print(err.Error())
        return "Failed post", err
    }
    return "Successfully posted", nil
}

func DeleteStock(event functions.Event, context functions.Context) (string,
error) {
    event.Extensions.Request.ParseForm()
    id := event.Extensions.Request.FormValue("id")

    stmt, err := db.Prepare("DELETE FROM stock where id= ?;")
    if err != nil {
        fmt.Print(err.Error())
    }
    _, err = stmt.Exec(id)
    if err != nil {
        fmt.Print(err.Error())
        return "Failed deleted", err
    }
    return "Successfully deleted", nil
}
```

Listing 2. Example code of FaaS functions

Above Listing 2 is example code of FaaS functions regarding stock. In Listing 2, top function PostStock creates new stock in the database using event.Data which contains data from request. Bottom function DeleteStock deletes stock in the database based on id. For acquiring id of stock to delete from request, event.Extensions.Request.ParseForm() is to parse request form and event.Extensions.Request.FormValue(“id”) is obatining value of id from request parameters.

4 Results

Chapter 4 aims to benchmark implementation of the paper, by analyzing existing and proposed solution. Benchmarking was processed on local machine with open source testing tool. The benchmarking was carried out in three phases. First phase was about CPU and memory usage of each solution during API call. Second phase was testing backend performance of each solution. In second phase, detailed endpoint performances were analyzed to demonstrate how each architecture was reliable with different variables. In addition, the second phase benchmarked the speed of same endpoint performance with difference variables to discover how different variables can influence. Finally, third phase was regarding development cycle of each solution.

To benchmark each architecture backend, backend performance testing was conducted by k6 testing tool. k6 is open source testing tool for testing backend performances, built with Go and JavaScript [18]. k6 testing tool provides multiple options for testing such as virtual user, duration of testing, object for third party collectors, hosts, iterations and so on. Among k6 options, virtual user and duration options were used to measure backend performance of FaaS and Microservices.

4.1 CPU and memory usage

Chapter 4.1 analyze CPU and memory usage of Microservices and FaaS. The data of Microservices and FaaS was collected with certain conditions. The conditions were same GET endpoints of each solution to get all books and the number of virtual users is ten users with different duration time. For collecting the data, Kubernetes dashboard and Heapster were used. Heapster was used for monitoring compute resource usage of clusters in Kubernetes. Heapster deployed in Kubernetes cluster and collected data from pod of each solution [19].

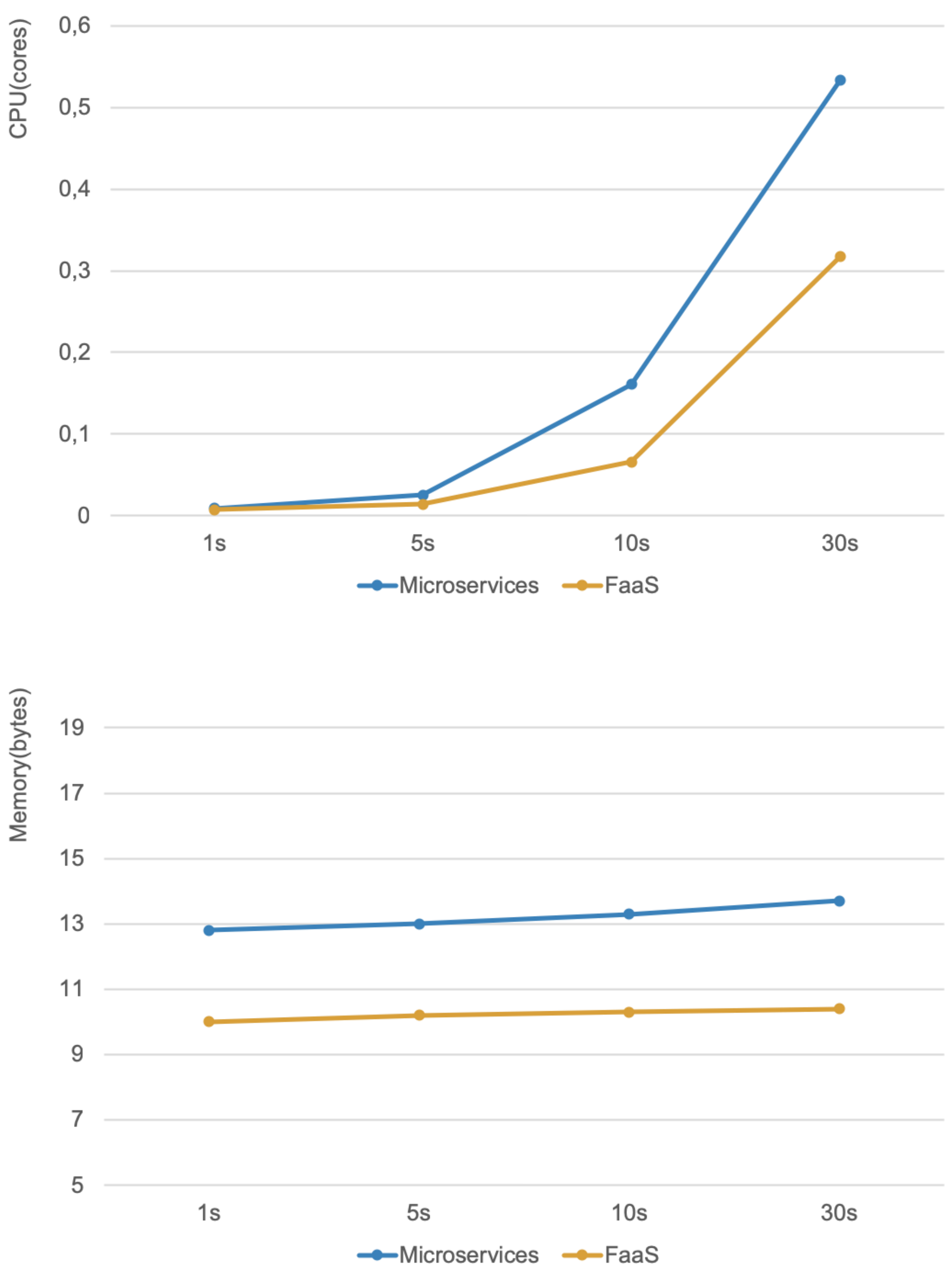


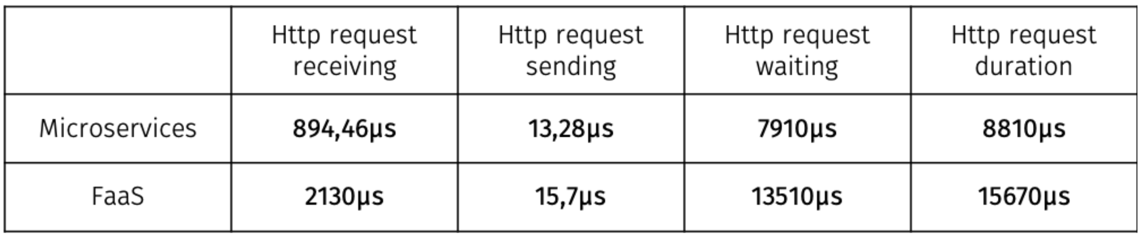
Figure 12. CPU and memory usage of Microservices and FaaS

Above Figure 12 illustrates CPU and memory usage of Microservices and FaaS. Above graph is CPU usage of Microservices and FaaS. Below graph is memory usage of

Microservices and FaaS. Overall, duration time increased and CPU and memory usage increased in Microservices and FaaS. The CPU usage of Microservices increased more rapidly and higher than the CPU usage of FaaS. As a result, Microservices consumed more CPU than FaaS. The memory usage of Microservices was higher than the memory usage of FaaS. Consequently, FaaS consumed less memory than Microservices while endpoint called. On the other hands, the efficiency of CPU and memory usage was better in FaaS than Microservices.

4.2 Backend performance testing

The chapter 4.2 focused on analyzing factors in terms of capability to handle endpoint. In addition, among endpoints, GET endpoint of Microservices and FaaS was chosen for basis of the testing.



	Http request receiving	Http request sending	Http request waiting	Http request duration
Microservices	**894,46µs**	**13,28µs**	**7910µs**	**8810µs**
FaaS	**2130µs**	**15,7µs**	**13510µs**	**15670µs**

Table 2. Microservices and FaaS http request metrics

Above Table 2 reveals Microservices and FaaS http request metrics in terms of http request duration, http request receive time, http request send time, and http request wait time. The conditions of Table 2 metrics were five virtual users, 30 seconds duration of testing, and GET request of each architecture. While in Microservices http request receiving was 894,46 microseconds, http request send time was 13,28 microseconds, and http request waiting was 7910 microseconds, in FaaS http request receiving was 2130 microseconds, http request send time was 15,7 Microservices, and http request waiting was 13510 microseconds. Total http request duration of Microservices was 8810 microseconds and FaaS was 15670 microseconds. Glancing overall http performances, Microservices was faster than FaaS in terms of http request receiving, http request sending, and http request waiting. As a result, http request of Microservices backend was 6860 microseconds faster than http request of FaaS backend.

Next benchmark was carried out with four cases to validate which variable can affect to API performance. First case was five virtual users and thirty seconds duration. Second case was five virtual users and ten seconds. Third case was one virtual user and thirty seconds. Final case was one virtual user and ten seconds.

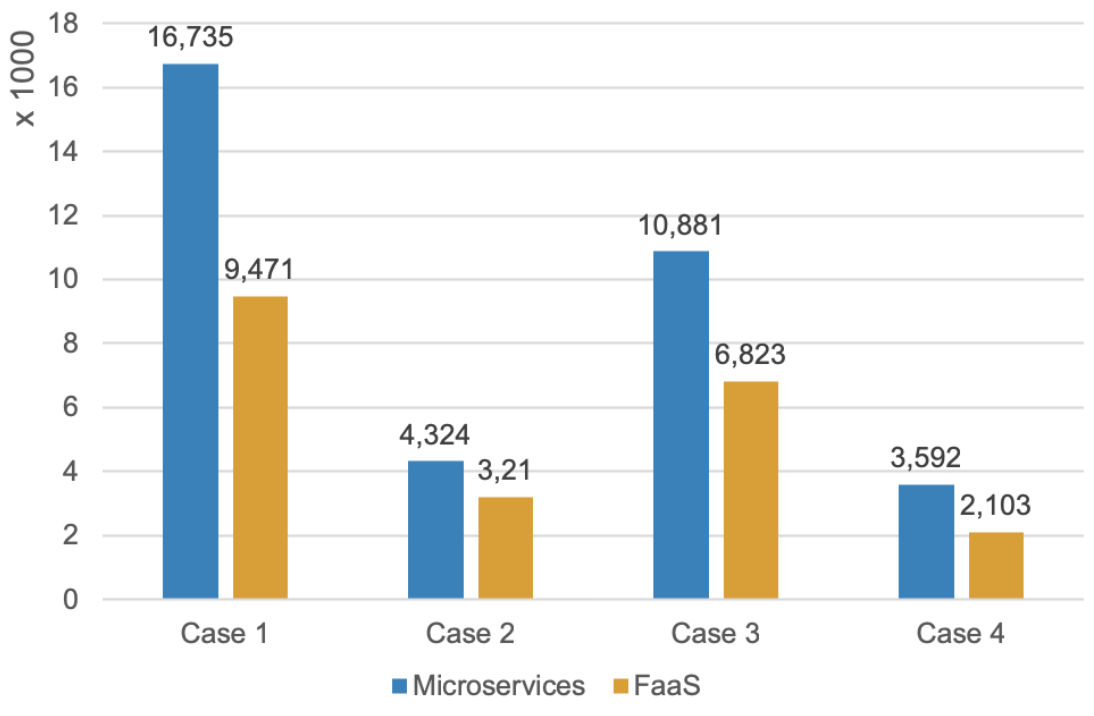


Figure 13. The number of successful http request

Above Figure 13 illustrates the number of successful http request by four cases. Overall, the number of Microservices successful http requests was more than the number of FaaS successful http requests. With same number of users and duration time, Microservices obtained more successful http requests than FaaS. Furthermore, in case 1 and case 3, the gap between Microservices and FaaS was larger than in case 2 and case 4. Consequently, the more users could cause less successful http request in FaaS.

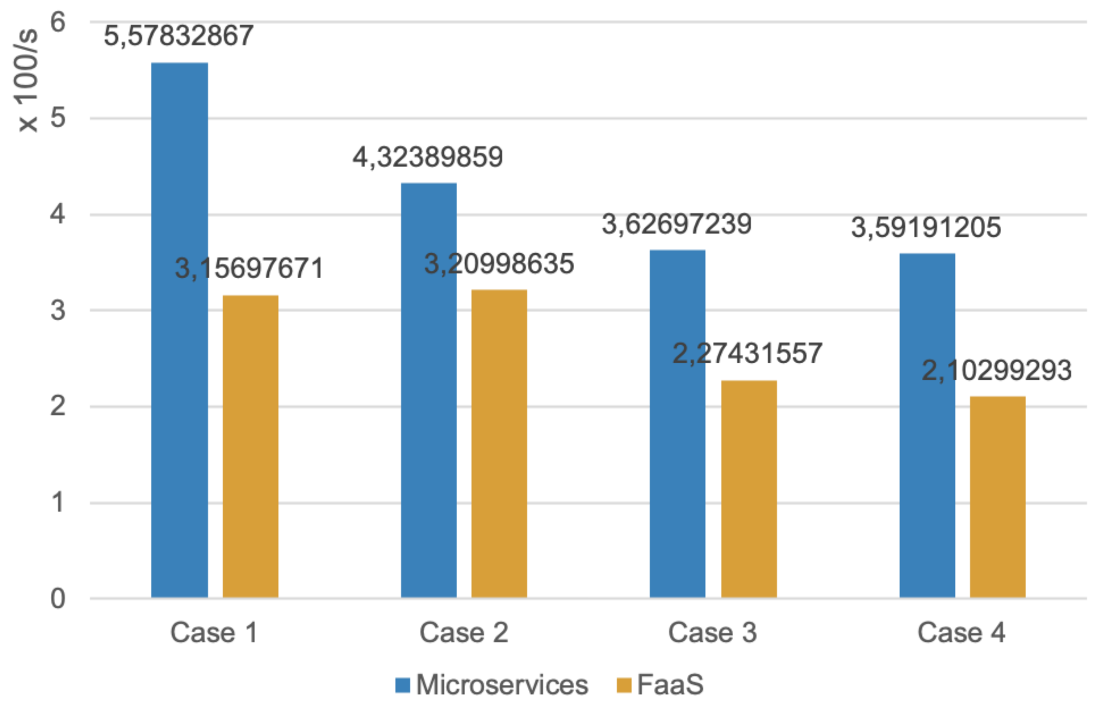


Figure 14. The velocity of each case

Above Figure 14 demonstrates the velocity of each case in Microservices and FaaS. The velocity was measured by the number of successful http requests divided by each case's duration. In all cases, the speed of Microservices backend was faster than FaaS backend. As a result, Microservices backend had higher capability to handle request per second than FaaS backend. Furthermore, the difference between velocities of Microservices in case 3 and 4 were slight, which means when a user sent http request, time did not affect Microservices. The difference between case 1 and 2 were larger, which means when five users sent http request, time could affect Microservices. As a result, time could make Microservices slow down when several users send http requests. On the other hand, the differences of FaaS velocities were subtle between case 1 and 2 and between case 3 and 4. However, grouping case 1 and 2, and case 3 and 4, the gap of two groups were comparatively larger. Therefore, the performance of FaaS was not affected by same number of user but could be influence by time.

4.3 Development cycle

Chapter 4.3 compared development cycle between Microservices and FaaS because FaaS changed the development cycle compared to standard development cycle. Below Figure 15 illustrates development cycle of each architecture.

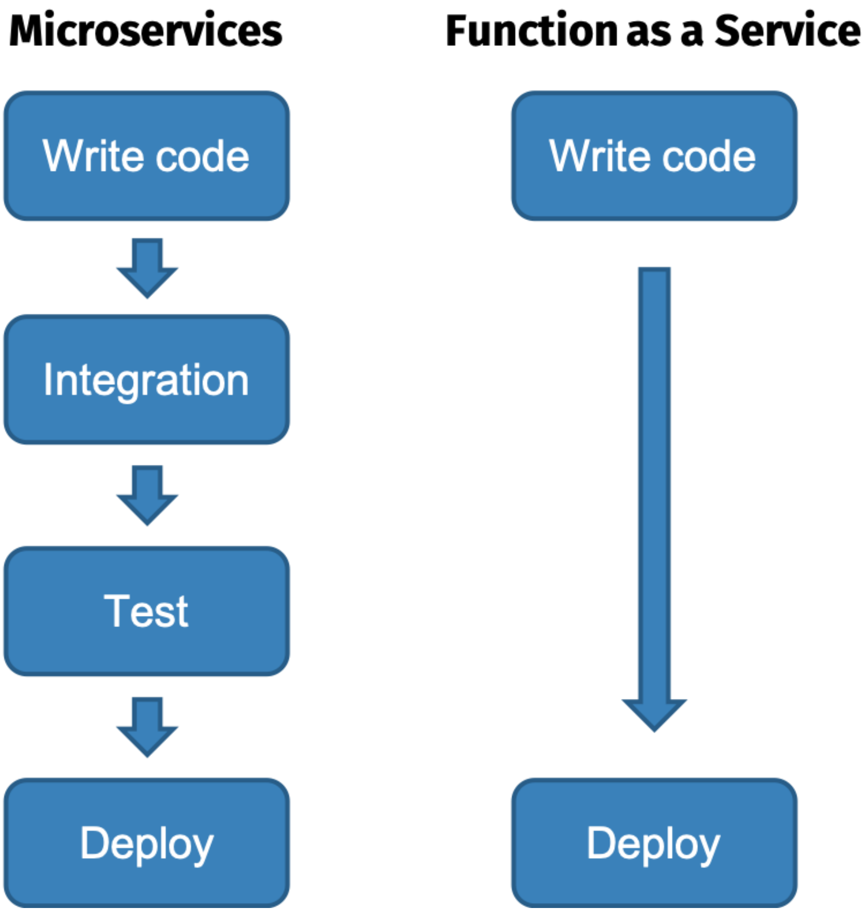


Figure 15. Development cycle of each architecture

In above Figure 15, Microservices comprises four steps such as write code, build, test, and deploy. In Microservices' development cycle, endpoint developers write codes. Next step is integrating the code changes into the existing functionalities. Then, endpoint developers write docker file and create docker image for a service which contains the change. After that, docker image with the changes are tested in Kubernetes to verify the changes works properly with existing codes in Kubernetes. When passing the test, the change can be deployed. On the other hands, FaaS comprises two steps such as write code and deploy. Endpoint developers write code and deploy it on the cloud. Assuming that the period of each steps is same, FaaS could be twice faster than Microservices. As a result, development cycle of FaaS is shorter than development cycle of Microservices.

However, deployment step using Kubeless framework was not reasonable. Kubeless has a build process for functions to create docker images for functions and store in a docker registry. When a function was deployed first time, minimum 3 minutes were taken until a function was deployed and running in Kubernetes. After the build process was completed, the function was ready to be invoked. This issue is known as cold start of FaaS. If cold starts happen often, FaaS is costly. On the other hands, Microservices' deployment run constantly so that Microservices does not have any issues similar with cold carts in FaaS.

5 Discussion

In the chapter 5, the goal is to evaluate the results of the paper and process of the implementation. In the discussion section, limitations from the results and the process of deriving the results and contributions of the results are mentioned.

5.1 Limitations and Weaknesses

In the entire process of thesis, some of limitations and weaknesses were present. First, the results of backend performance were different with the expected result that Microservices combined with FaaS could improve backend performance. As briefly mentioned about the results, the number of successful http requests in Microservices was more than FaaS. In addition, in spite of different variables, the velocity of http request in Microservices was faster than FaaS. As a result, Microservices had higher stability and availability than FaaS in terms of handling endpoint. In other words, the existing solution performed more effectively than the proposed solution. Some of factors could influence the result. First factor could be not stable environment to test. Prototype application was running on local machine, not on the cloud. The running environment for the prototype can be distracted by different jobs, for example background applications and background programs on local machine. Therefore, the running environment could prevent that the prototype worked solidly. Secondly, open source framework had limitations to implement FaaS. Open source framework is developing and is not fully supporting functionalities for FaaS such as auto-cleaning function in the container after the execution of function completed. Consequently, the fully implementation of FaaS was difficult with open source framework. Final factor was the prototype size. The prototype was small and simplified, thus the size was not adequate size for actual enterprise application. Big-size application could show more reliable results than small-size application.

Second was slow build process of Kubeless framework. Kubeless framework has complex build process to deploy functions. The build process caused cold start. For deploying a function, minimum time was 3 minutes so that the build process slowed down deployment process for functions. Therefore, first invocation of FaaS functions were especially slow and remained room for further improvement.

Third limitation and weaknesses were in terms of resources about FaaS. The period of researching FaaS is shorter than the period of researching Microservices. Therefore, FaaS resources were comparatively smaller than Microservices resources. Most of resources contained similar and shallow information, not deeply theoretical information. In the process of literature review, collecting valuable information about FaaS was challenging. Moreover, the shortage of FaaS resources caused problems in implementation and result steps. Especially, testing tools and open source frameworks for FaaS were few and still working in progress. In addition, setting up the local environment for each solution and finding proper tools for the testing were difficult in a short term. Therefore, the limitation caused to increase the amount of efforts for the testing.

5.2 Contributions and Achievements

Even though limitations and weaknesses were existed, there were contributions and achievements from the paper. First was efficiency of CPU and memory usage in FaaS. Due to small unit called a function, FaaS used less CPU and memory than Microservices with different conditions. As a result, FaaS provided more economical CPU and memory usage than Microservices.

Second was fast development cycle of FaaS. FaaS decreased four steps of traditional development cycle to two steps of development cycle. Traditional development cycle is writing code, integrating, testing, and deploying. On the other hand, FaaS development cycle is writing code and deploying. Therefore, FaaS development cycle achieved shorter development cycle, saving time and efforts for endpoint developers to integrate and test.

In addition, the paper could contribute to determine application architecture for enterprises or personal project in the future. FaaS is not settled technology but growing and developing rapidly. The needs to integrate FaaS into businesses are growing and the interest of FaaS efficiency is increased. This paper provided information about analyzing actual performance such as CPU and memory usage, reliability, and velocity to help decision making depending on their use cases and requirements.

In the whole process of thesis, researching application architecture deeply, planning implementation and testing, and conducting them were great achievement personally. From the literature review, achievement was to obtain theoretical backgrounds of Microservices and FaaS. Theoretical backgrounds from the literature review was actualized in methods and materials. In the results, the achievement was learning practical testing via benchmarking the project implemented in methods and materials. Although the result was different with the expected, seeking the reasons of the result were valuable time to learn and understand more.

6 Conclusion

The goal of this paper was to examine how FaaS can be improved backend by incorporating FaaS into Microservices architecture. Through the entire process, this paper obtained some of drawbacks and benefits. Firstly, backend performance testing indicated Microservices had higher stability and availability than FaaS. In addition, the impact of cold start in FaaS was sizable due to Kubeless framework' build process. Moreover, lack of tools, frameworks, and other resources affected to collect information, implement, and test because FaaS is relatively new technology. On the other hands, CPU and memory efficiency was higher in FaaS than Microservices and development cycle in FaaS was shorter than Microservices. Moreover, the paper granted information about actual implementation and benchmarking Microservices and FaaS, thus the information could contribute decision making about application architecture.

To conclude, both architectures had benefits and disadvantages. FaaS did not indicate better performance of backend than Microservices. However, FaaS was efficient in terms of resource consumption and showed the possibility of improving development environment. As a result, the decision between Microservices and Microservices combined with FaaS is needed to consider the use case. This paper remained some of improvements in both application architecture. Therefore, this paper suggests for further researches that improving low capability of monitoring and debugging and cold start in FaaS.

References

1 Rashmi Rai, Gadadhar Sahoo, Shabana Mehfuz (2015) Exploring the factors influencing the cloud computing adoption: a systematic study on cloud migration [e-journal]. SpringerPlus; April 2015, 197. URL: https://doi.org/10.1186/s40064-015-0962-2.

2 Sam Newman (2015) Building Microservices [ebook]. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc.; February 2015. URL: https://books.google.fi.

3 Daniel Richter, Marcus Konrad, Katharina Utecht, Andreas Polze (2017) Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice. IEEE; July 2017. URL: https://ieeexplore.ieee.org/document/8004304/authors#authors.

4 Brian Zambrano (2018) Serverless Design Patters and Best Practices: Build, secure, and deploy enterprise ready serverless applications with AWS to improve developer productivity [ebook]. Packt Publishing Ltd; April 2018. URL: https://books.google.fi.

5 Kuldeep Chownhan (2018) Hands-on Serverless Computing: Build, run and orchestrate serverless applications using AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions [ebook]. Packt Publishing Ltd; July 2018. URL: https://books.google.fi.

6 Barrie Sosinsky (2011) Cloud Computing Bible [ebook]. Wiley Publishing, Inc., Indianapolis, Indiana; January 2011. URL: https://metropolia.finna.fi/Record/nelli15.3390000000007759.

7 Alexa Huth, James Cebula (2012) The Basics of Cloiud Computing. US-CERT; June 2012. URL: https://www.us-cert.gov/security-publications/basics-cloud-computing.

8 Miika Kalske (2017) Transforming monolithic architecture towards microservice architecture. University of Helsinki; April 2018. URL: https://helda.helsinki.fi/handle/10138/234239.

9 Nicola Dragoni, Ivan Lanese, Stephan Larsen, Manuel Mazzara, Ruslan Mustafin, et al.. Microservices: How To Make Your Application Scale. Moscow, Russia: *A.P. Ershov Informatics Conference (the PSI Conference Series, 11th edition)* [online]; Jun 2017. <hal-01636132>, version 1. URL: https://hal.inria.fr/hal-01636132.

10 Ghofrani, Javad and Daniel Lübke (2018). Challenges of Microservices Architecture: A Survey on the State of the Practice. *ZEUS*; 2018. URL: https://www.semanticscholar.org/paper/Challenges-of-Microservices-Architecture%3A-A-Survey-Ghofrani-Lübke/adf96f220a1761f8c5fee219d169d725ea7bedbd.

11 Peter Sbarski (2017). Serverless Architectures on AWS: With examples using AWS Lambda [ebook]. Manning Publications Co; April 2017. URL: https://metropolia.finna.fi/Record/nelli15.3790000000320818.

12 Mohamed Labouardy (2018). Hands-On Serverless Applications with Go: Build real-world, production-ready application with AWS Lambda [ebook]. Packt Publishing Ltd; August 2018. URL: https://books.google.fi.

13 Diego Zanon (2017) Building Serverless Web Applications: Build scalable web apps using Serverless Framework on AWS [ebook]. Packt Publishing Ltd; July 2017. URL: https://books.google.fi.

14 Shashikant Bangera (2018) DevOps for Serverless Applications: Design, deploy, and monitor your serverless applications using DevOps practices [ebook]. Packt Publishing Ltd; September 2018. URL: https://books.google.fi.

15 Kubeless (2018) Kubeless; July 2018. [cited on 1 Nov 2018]. URL: https://github.com/kubeless/kubeless.

16 OpenFaaS – Serverless Functions Made Simple (2017) OpenFaaS; Nov 2017. [cited on 3 Nov 2018]. URL: https://github.com/openfaas/faas.

17 Build process for functions. Kubeless 2018 [cited on 1 Nov 2018]. URL: https://kubeless.io/docs/building-functions.

18 k6. k6; Nov 2018. [cited on 12 Nov 2018]. URL: https://docs.k6.io/docs.

19 Heapster. Kubernetes; May 2018. [cited on 15 Nov 2018]. URL: https://github.com/kubernetes/heapster.