Aalto University
School of Science
Master's Programme in ICT Innovation

Muhammad Shoaib Khan

Scalable invoice-based B2B payments with microservices

Master's Thesis
Espoo, December 31, 2019

Supervisors: Professor Mario Di Francesco, Aalto University
Advisor: Gopika Premsankar M.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in ICT Innovation

ABSTRACT OF
MASTER'S THESIS

Author:	Muhammad Shoaib Khan		
Title:			
Scalable invoice-based B2B payments with microservices			
Date:	December 31, 2019	**Pages:**	70
Major:	Cloud Computing and Services	**Code:**	SCI3081
Supervisors:	Professor Mario Di Francesco		
Advisor:	Gopika Premsankar M.Sc. (Tech.)		

Paying by invoice has several advantages for businesses over conventional payment methods such as Debit/Credit cards. An invoice not only allows a buyer to make the purchase on credit but also contains the tax information for each purchased item. Businesses need to save this information in their financial records and report it to the authorities. The core challenge in an invoice-based payment method is the ability to make an accurate credit decision for a given purchase. Such a credit decision requires information about the buying company such as their credit rating. The company information is gathered in real time from different third-party sources. In this context, Enterpay Oy provides an invoiced-based B2B payment solution and is growing its payment service to European countries. In order to support this expansion, Enterpay needs to develop new capabilities such as the ability to detect fraudulent purchases. These new features require the application architecture to be flexible in terms of technology. For example, different components of the service should be built with the most suited programming language, libraries, and frameworks.

The goal of this thesis is to enable efficient scaling and high availability for Enterpay's payment service. Thus, we have migrated from a monolithic a microservice-based architecture. This transition allows us to choose the best suited technology for the business case of the given microservice. We extracted various modules from the original monolithic application, which have different scalability criteria. We built these modules as Docker containers, which run as independent microservices. We used Kubernetes as the container orchestration framework and deployed the microservice in Amazon Web Services (AWS). Finally, we conducted experiments to measure the performance of the service with the new architecture. We found that this architecture not only scales faster but also recovers from instance failures quicker than the previous solution. Additionally, we noticed that the average response time of service request is similar in both architectures. Finally, we observed that new microservices can be built using different technology stack and deployed conveniently in the same Kubernetes cluster.

Keywords:	B2B payments, Docker, Kubernetes, Monolithic architecture, Microservices, containers, virtualization
Language:	English

Acknowledgements

I would like to extend my gratitude to my supervisor Professor Mario Di Francesco and the instructor Gopika Premsankar for their continuous support and guidance throughout the course of this thesis. I would like to express my deepest appreciation to Jarkko Anttiroiko and Tapio Vepsäläinen for providing me related business insights of the case study and allowing me to write this thesis during the official working hours. Finally, I would like to thank my wife for her patience and continuous encouragement throughout this thesis. Writing this thesis would have been impossible without the help and support of above mentioned people.

Thank you.

Espoo, December 31, 2019

Muhammad Shoaib Khan

Abbreviations and Acronyms

AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
B2B	Business-to-Business
CD	Continuous Deployment
CI	Continuous Integration
CLI	Command Line Interface
DRY	Do not Repeat Yourself
EC2	Elastic Compute Cloud
ECS	Elastic Container Service
ELB	Elastic Load Balancer
IDE	Integrated Development Environments
JAR	Java ARchive
JVM	Java Virtual Machine
OTP	One Time Password
PSP	Payment Service Provider
RDBMS	Relational Database Management System
RDS	Relation Database Service
SRP	Single Responsibility Principle
SSH	Secure Shell
UI	User Interface
URL	Uniform Resource Locator
VAT	Value Added Tax
VM	Virtual Machine
VPC	Virtual Private Cloud

Contents

Abbreviations and Acronyms **4**

1 Introduction **8**
1.1 Business case 9
1.1.1 Problem statement 9
1.2 Contribution 10
1.3 Structure of the thesis 11

2 Virtualization **12**
2.1 What is virtualization? 12
2.1.1 Hypervisor-based virtualization 14
2.1.2 Container-based virtualization 15
2.2 Docker 16
2.2.1 Docker engine 17
2.2.2 Docker architecture and objects 17

3 Software Architecture **21**
3.1 Monolithic architecture 21
3.2 Microservice-based architecture 24
3.2.1 Software containers in microservice-based architecture . 27

4 Container Orchestration **29**
4.1 What is container orchestration? 29
4.2 Kubernetes 30
4.2.1 Master 31
4.2.2 Pods 32
4.2.3 Node Server 33
4.2.4 Replication Controller 34
4.2.5 Deployment 34
4.2.6 Services 35
4.2.7 Ingress 35

4.2.8 Web User Interface 35
4.3 Elastic Container Service (ECS) 35
4.3.1 Task Definition 36
4.3.2 Task 36
4.3.3 Scheduler 36
4.3.4 Cluster 36
4.3.5 ECS container instance 37
4.3.6 Container Agent 37
4.3.7 Service 38

5 Invoice-based B2B payments 39
5.1 What is B2B payment? 39
5.2 Enterpay's invoice-based B2B payment Service 40
5.2.1 Life cycle of a purchase 41
5.2.2 Benefits of invoices for businesses 42
5.2.3 Features of the payment service 43

6 Implementation 45
6.1 Migration to microservice-based architecture 45
6.1.1 Introducing Docker 45
6.1.2 Modularization - Transition to Microservices 46
6.1.3 Deploying to Kubernetes cluster 48

7 Evaluation 51
7.1 Experimental setup and methodology 51
7.1.1 Hypothesis and metrics 53
7.2 Experimental results 53
7.2.1 Time to recover 53
7.2.2 Time to scale 54
7.2.3 Response time for service requests 55
7.3 Considerations on flexibility 56

8 Conclusions 58
8.1 Future work 59

A Implementation - Intermediate steps 67
A.1 docker-compose 67
A.2 Deployment on local Kubernetes cluster 68
A.3 Deployment on AWS Kubernetes cluster 69
A.4 Kubernetes cluster resources 70

List of Figures

2.1 Overview of how virtualization enables running multiple applications targeting multiple operating systems on the same hardware [21] 13
2.2 Overview of how virtualization enables running multiple applications targeting multiple operating systems on the same hardware [47] 14
2.3 Virtual machines vs Containers [18] 16
2.4 Docker architecture [12] 18

3.1 Typical layered architecture of monolithic applications 22
3.2 High level view of microservices based application 25

4.1 Overview of the various components in Kubernetes architecture [33] 34
4.2 Overview of Amazon Elastic Cloud Service 37

5.1 Overview of the life cycle of an invoice-based B2B purchase . . 42

6.1 Overview of how ingress controller directs the traffic to different services 50

7.1 Experimental setup for the monolithic architecture 52
7.2 Average time to recover from instance failure(s) 54
7.3 Average time to scale out given number of instances 55
7.4 Average response time for concurrent service requests 56

Chapter 1

Introduction

The traditional software development approach aims to build a complete application as a single unit. All of the modules or components in an application are packaged together in a single deployment artifact [39]. Although this strategy is easy and convenient option to start with, it can become an obstacle for scalability [43]. This development methodology makes the application rigid and difficult to change in the future. In practice, many software applications face the challenge of adapting to the dynamics of growth and agility. These applications typically undergo architectural changes to match the scaling requirements and to provide new features and services to their end users. Applications failing to meet these requirements, not only risk loosing the competitive advantage, but can quickly become obsolete.

In general, web applications can be deployed either on premises or over the cloud. The Cloud computing is an efficient solution for deployment as the cloud provider takes the responsibility of management, configuration, and maintenance of the server hardware. Cloud providers rely heavily on virtualization, which allows different applications targeting different operating systems to run simultaneously on a given computer system [52]. These operating systems run in a layer abstracted from the actual hardware in isolation [47]. Virtualization enables efficient sharing of resources across multiple operating systems.

Software containers aim to implement the virtualization logic at the operating system level [48]. It packages an application along with its dependencies and runtime environment in a single deployment unit. Each application runs inside a container and one can create multiple copies of the application to scale out when required. Docker is a famous containerization platform, which provides en efficient way of packaging the application and dependencies inside a docker image [11]. This image can be saved in a local or remote repository.

The microservice-based architecture is a modern approach for software de-

velopment. It focuses on building an application as a group of many loosely coupled services, which run independently in a a distributed system. These services communicate with one another by lightweight communication mechanisms, such as Application Programming Interface (API) calls over HTTP [41]. Different microservices running in an application can have different scalability needs. Thus, one microservice can be scaled independently according to its own needs. Containerization is best suited for microservices applications [57]. Each microservice can run as a container and use minimum resources.

Container orchestration is essentially the process of managing the life cycle of the containers, particularly in large and complex environments. Kubernetes is a platform for container orchestration, which was originally developed by Google [59]. It provides an effective deployment and management of container's life cycle in clusters. Kubernetes provides services that are helpful to achieve predictability, scalability, and availability [25].

1.1 Business case

Many businesses rely on the goods, and services offered by some other business to produce their own goods and services. Such transactions where both the seller and the buyer are businesses, are termed as Business-to-Business (B2B) trade. The conventional payment methods such as Checks and Debit/Credit cards are inefficient for B2B trade. Paying with these methods is not only cumbersome but also inconvenient when it comes to keeping financial records up-to-date. On the contrary, paying by an invoice is a preferred by businesses because it solves the problems posed by the conventional payment methods. For example, invoices allow the buyers to purchase the items on Credit and offer a convenient way to update the financial records.

Enterpay Oy provides an invoice-based B2B payment method, which enables the buyers to make purchases on credit and takes the credit risk away from sellers. From the prospective of banks and financial companies, Enterpay offers an online credit decision engine that helps them expand their business and offer a reliable payment service in the B2B market.

1.1.1 Problem statement

In recent times, Enterpay has been growing its payment service to various European countries. Enabling the payment service in a new country, requires localization of the application for that particular country. This localization typically involves an integration with local information providers, creating a

credit policy for the given country, and implementing some country-specific features. Each localization introduces complexity in the application. Furthermore, the payment service needs to be highly available in different countries and should scale efficiently when required.

Additionally, Enterpay wants to develop new capabilities such as the ability to detect fraudulent purchases. These new features require different components of the service to be built with the different programming languages, libraries, and frameworks. For instance, fraud detection requires implementing different machine learning algorithms. These algorithms can be conveniently implemented in the Python programming language due to the availability of many libraries. Consequently, the application architecture needs to be flexible in choice of technology.

This thesis has the following two primary objectives

- Enable high availability and efficient scalability for Enterpay's payment service.
- Introduce flexibility in the application architecture, so that the new features can be built with the best suited programming language, libraries and frameworks.

1.2 Contribution

We proposed microservice-based architecture as the solution for the abovementioned challenges.

- We identified 5 modules of the original monolithic application that could be extracted and run independently as microservices. Each microservice was built as a separate Docker image.
- For each microservice, we created kubernetes deployment resources and defined replication strategy. Additionally, we created service resources for load balancing between multiple pods of the individual microservice. Finally, we added an ingress resources to implement the API gateway design pattern.
- We conducted experiments to measure the performance of microservice-based architecture. We noticed that the new microservice-based architecture offers better scaling and recovers from instance failures faster than the previous solution.
- The new microservices targeting different technology stack were successfully deployed in the same Kubernetes cluster.

1.3 Structure of the thesis

Chapter 2 introduces the background technical concepts such as virtualization and container-based virtualization. Additionally, it discusses the prevalent platform for software containers ,i.e, Docker. Chapter 3 compares two software development methodologies ,i.e, monolithic and microservice-based architecture. Chapter 4 explains the process of container orchestration. It also provide a detailed discussion on Kubernetes, the most widely used container orchestration framework. Chapter 5 introduces invoice-based B2B payments. Additionally, it presents the payment service of Enterpay, which serves as the case study for this thesis. Chapter 6 provides implementation details of the migration from monolithic to microservice-based architecture. Chapter 7 evaluates the microservice-based architecture. Finally, Chapter 8 concludes this thesis.

Chapter 2

Virtualization

This chapter introduces the fundamental technologies that are related to this thesis. It first discusses the concept of virtualization and analyzes hypervisor-based virtualization and container-based virtualization. Finally, it studies Docker, the prevalent tool for software containers, and reviews its architecture in detail. The discussion presented in this chapter serves as the foundation for the concepts and tools explained in next chapters.

2.1 What is virtualization?

In general, virtualization means running a virtual instance of an operating system on an actual hardware. Virtualization allows different operating systems to run in parallel on a computer system. From the perspective of running operating systems, each operating system thinks that it is running on a dedicated hardware [47].

In other words, virtualization consolidates the physical resources of a hardware as a logical pool of resources, and allows sharing of these resources across multiple operating systems that are running on the same hardware. This resource sharing is done via different techniques including hardware/-software partitioning, emulation etc [19]. These virtual operating systems are often referred to as virtual machines and are run in isolation from one another. These virtual machines share resources like processor, storage etc depending on their needs, thus reduce the hardware cost and yield better resource utilization. Figure 2.1 shows how various virtual machines can be run on a single physical host using virtualization.

Virtualization allows the scaling of resources. For example, when the demand increases, these virtual machines can be scaled horizontally by running more virtual-instances/replicas. However, this setup would require a

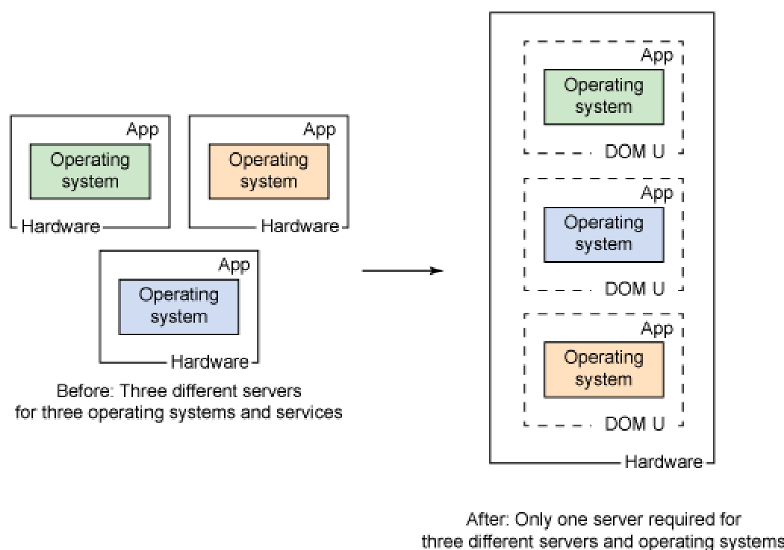

Figure 2.1: Overview of how virtualization enables running multiple applications targeting multiple operating systems on the same hardware [21]

load balancer, which will be placed in front of these replicas. The load balancer will receive outside traffic and distribute it amongst these replicas. On the other hand, scaling a virtual machine vertically involves allocating more resources to a given virtual machine. Similarly, when the demand reduces, additional resources can be taken back or additional instances can be terminated as well [2].

Additionally, virtualization allows each virtual machine to run its own operating system in complete isolation from other virtual machines and the host operating system. Consequently, different virtual machines with heterogeneous operating systems can run simultaneously on the same hardware [52].

Another advantage of virtualization is that it facilitates consistency among development, testing and production environments. It allows a developer to emulate the production/testing environment on his/her own computer by running a virtual machine matching the production/testing environment. This enables the developer to test the application behavior against the desired environment and reduces the risk of unwanted deployment issues when the application is released to production.

Below we will briefly discuss the two popular virtualization types.

2.1.1 Hypervisor-based virtualization

The Hypervisor is a software component that controls the system resources and administers the sharing of resources among virtual machines. The hypervisor is responsible to create virtual machines and manage their assigned resources. From virtual machine's perspective, each Virtual Machine (VM) gets a run-time environment which is identical to the dedicated hardware.

Hypervisors are generally classified in two types [49]. The first kind is called *Type-1 or native/bare-metal hypervisors* and the second kind is called *Type-2 or hosted hypervisors.*

- **Type-1** hypervisor operates directly on the host hardware and administers the virtual operating systems. An example of commercially used type-1 hypervisor is Microsoft's Hyper-V.
- **Type-2** functions as a process running on the host operating system abstracting the host operating system from virtual machines. An example of commercially used type-2 hypervisor is VMware WorkStation.

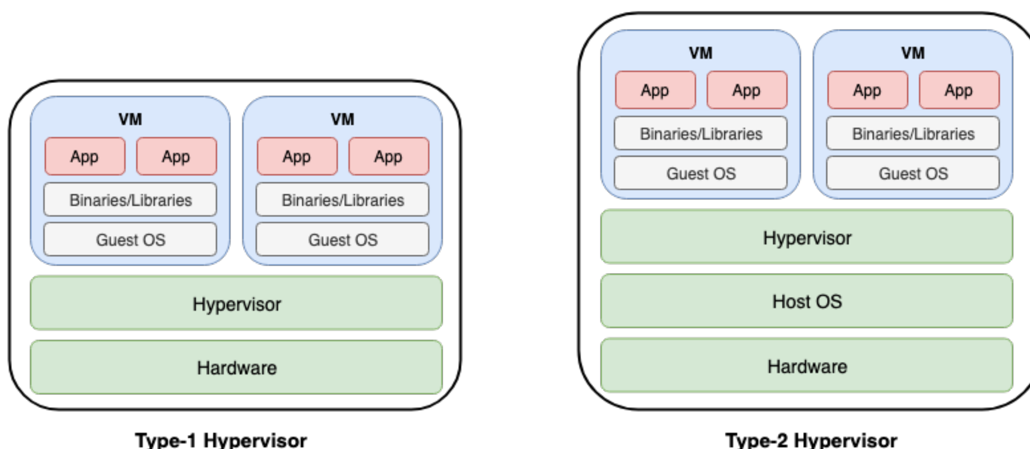

Figure 2.2: Overview of how virtualization enables running multiple applications targeting multiple operating systems on the same hardware [47]

As shown in Figure 2.2, type-2 hypervisors coordinate with the host operating system to serve the resources required by guest operating systems. This makes them a good choice for running multiple virtual machines on a single personal computer. In contrast, type-1 hypervisors are directly installed on dedicated hardware and have a management console, which means that they are more likely to be used in data centres [37]. The choice of hypervisor type depends on the performance metrics like CPU overhead, memory, number of supported host operating systems etc.

Irrespective of the hypervisor types, virtual machines provide strong isolation [17]. A problem in any of the guest operating systems does not usually affect the host operating system or the other guest operating systems running on the same hardware. However, this isolation comes at the expense of computational overhead required for virtualizing the hardware [7].

Furthermore, virtual machines can take a long time to get started as they require to boot the complete guest operating system, in addition to the software program they aim to run. Moreover, two software programs targeting the same operating systems, might have higher needs for isolation and require to be run in separate virtual machines. Consequently, running multiple virtual machines to support high isolation can result in inefficient resource utilization [7].

2.1.2 Container-based virtualization

Container-based virtualization or containerization provides a light weight substitute for hypervisor based virtualization. Containerization does not require a hypervisor and the virtualization is done on OS level, where multiple isolated user-space instances can coexist on the same kernel [48].

Containerization aims to encapsulate a given software application along with all of its dependencies and operating environment inside a software container. Bundling the application in a container provides abstraction from the actual run environment. Multiple containers running on the same host OS, will share the resources and can communicate with one another when required. Additionally, these containers are independent and isolated. However, this isolation is weaker when compared to hypervisor-based virtualization [46].

Unlike virtualization, one does not need to launch an entire virtual machine for every application, rather multiple containers can run within a single host [35]. This means instead of running multiple guest operating systems on the host operating system, multiple containers run directly on host operating system. This makes containers a lightweight substitute of virtual machines because they have low start time and require only a fraction of the memory as compared to virtual machines [18]. Similarly, containers ensure isolation between different programs by utilizing the low-level mechanics of host operating system. The cost of this isolation is considerably small when compared to that of virtual machines. Consequently, containers results in efficient resource utilization because of low overhead of virtualization of hardware resources [10].

Figure 2.3 compares the container based virtualization with the VM based virtualization. The container based infrastructure is running only one oper-

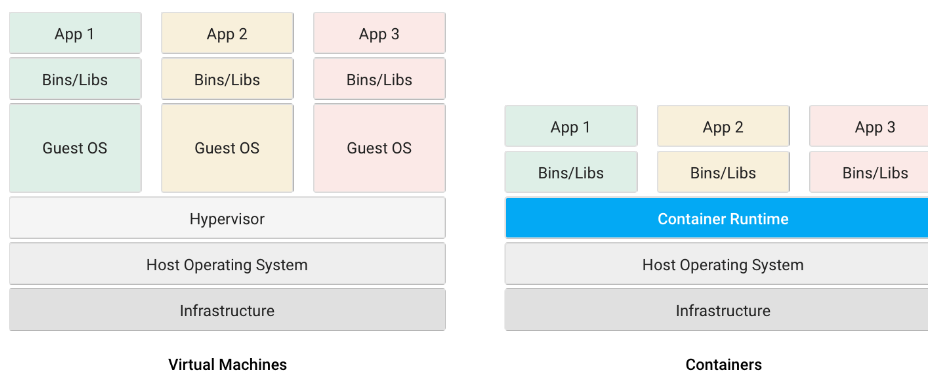

Figure 2.3: Virtual machines vs Containers [18]

ating system and there are three containers running independently on the same host operating system. The absence of guest operating system makes this scheme more resource efficient when compared to virtual machines.

Just like virtual machines, container are also built from an executable image. This image is a file which comprises of all essential pieces of information like the code, dependencies, run-time and system resources that are required to run the container.

2.2 Docker

Docker is a software program that provides a platform to develop, maintain and deploy applications inside containers [11]. Docker allows the developers to focus on development activities rather than the system on which the application will be deployed. Additionally, the developers can speed up the development by utilizing different tools that are already developed for docker environment. Docker helps to reduces the size of the application because some of the application dependencies can be resolved from the host system. This makes Docker beneficial for system admins too, as it provides flexibility with the choice of system and yields a smaller footprint with lower overhead [55].

Docker bundles all of the application dependencies and configurations in a single portable entity called docker image. This docker image is used to instantiate one or more containers. Docker containers can run on one or more hosts in isolation [12]. These containers are designed to be stateless so that they can be scaled up and down at any time to meet the application requirements.

Docker introduces efficiency in software development by allowing develop-

ers to create identical production environment on their development machine. The docker container, containing the necessary application dependencies and infrastructure details, can be run on developer's machine. The developer can essentially find and fix all the bugs that would otherwise be found only in the production environment. Similarly, containers are a good choice to implement continuous integration (CI) and continuous deployment (CD) work flows. Thus, docker allows development teams to move away from the conventional waterfall model and adapt the modern Agile practices of software delivery [16].

2.2.1 Docker engine

Before we analyze the architecture of docker, it is important to get the high level understanding of the docker. The docker engine follows the client-server architecture and allows the developers to develop, deploy and maintain applications with containers. It has the following three main components.

- **Docker Daemon.** It is the core component responsible for carrying out the docker jobs like creating and running containers etc. Docker daemon manages the resources such as storage and network, and runs as a background process. Docker daemon waits for the client request and executes it.

- **REST API.** This is an interface exposed to the client application to communicate with the daemon. Applications can use HTTP protocols to send their jobs to the daemon.

- **Docker CLI.** This is a command line interface (CLI) to communicate with the daemon. This interface allows developers to efficiently manage the container instances via direct commands.

2.2.2 Docker architecture and objects

The client-server model of docker has three tiers, i-e., docker client, docker host and the registry. Figure 2.4 elaborates the architecture of docker. The docker client is authorized to establish communication with the docker daemon, which otherwise is not accessible to the end user. The end user sends a command using docker client, which may not necessarily be on the same machine as the daemon. The command is received by the docker daemon via Rest API interfaces. Docker daemon then executes the command and sends back the response. The daemon is responsible for building, running, and distributing the containers. Additionally, the docker daemon can communicate

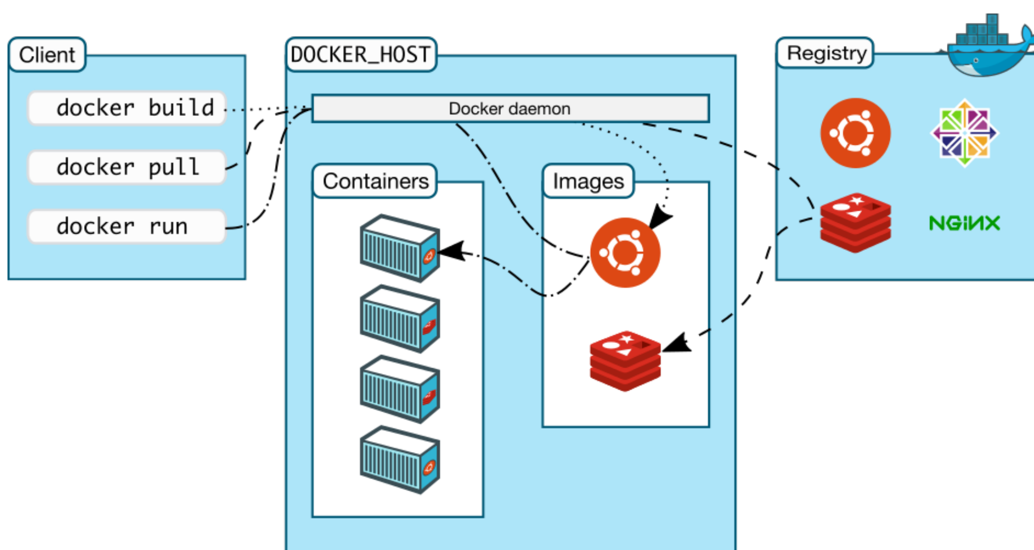

Figure 2.4: Docker architecture [12]

with other daemons and services to execute the required task efficiently. The docker architecture has the following entities called docker objects:

Docker Image

Docker image is a read-only binary template which is used to instantiate a docker container. This image is a portable entity that is used to store and ship the software module or the application. A docker image, in its own, contains all the information that is required to build the container, such as the system requirements and capabilities of the container[6]. A docker image is built on as the base image, which typically contains the system requirements. Further customization is performed on the image as per the needs of the application. For example, all of the libraries and other dependencies that are needed for the application to run, would be added in the docker image. These modifications create a new layer on top of the previous image. Each layer contains only the difference between the previous layer, thus allows the developers to track individual changes done in the docker image [8].

Docker Registry

Docker registries provide storage for docker images. In simple words, docker registries contain the repositories where one can store docker images and retrieve them when necessary. These registries can be public, where the image is stored to a remote location and the developer can manage its access

for outside world or his own peers. Some of the popular public registries are Docker Hub and Docker cloud. On the other hand, one can choose to store the images on an internal storage location of the organisation, which can be accessible over the intranet. The usage of these repositories is familiar for the developers as it is similar to the popular source code version control systems like GIT [23].

Dockerfile

The dockerfile consists of the instruction to automatically build the image. This file consists of the executable commands in the correct order. For example, a docker file contains the information where to pull the base image from and what configurations should be applied to prepare the required runtime for the application. The user can use the docker build command which can execute command-line instructions in sequence and yield an automated build.

Multi-stage Dockerfile

Managing the size of the docker image and ensuring that it stays within reasonable limits, can become demanding with the regular docker file. A docker file usually adds a layer with every command. All the artifacts that are not required in future layers, should be removed to keep reduce the image size. Consequently, in many scenarios, you would need to maintain different dockerfiles for different hosting environments such as development, testing and production. Each docker file would be optimized to have only the required libraries for the said environment. This approach is far from being ideal, as the multistage docker file provides a more efficient alternative. It offers the possibility to have multiple stages. Each stage can use a different base and one can choose to copy only the required artifacts from one stage to another. Thus, a single multistage docker file can yield the environment specific image for multiple environments [15].

Docker container

Containers are the instances built from the docker image [1]. A single docker image can be used to instantiate as many containers as required. Additionally, each container has a writable layer, which is used to maintain the state of the said container. The state of the container usually gets deleted when a container is terminated. However, this state can be persisted if required.

[1] https://www.docker.com/resources/what-container

The containers are run by using the *docker run* command. Applications and services are deployed inside a container and run in isolation from applications and services running in other containers on the same host.

Docker services

Services offer a convenient way to scale the application as per the needs. These services create a swarm of docker daemons, where each service can communicate to different daemons. User can define scalability configurations. For instance, the maximum/minimum container instances running in parallel. Additionally, services provide load balancing, which means that all the requests coming to the service are distributed across all running containers [12].

Chapter 3

Software Architecture

This chapter focuses on software architecture and discusses two prominent software design strategies. It aims to illustrate some of the benefits and challenges posed by these design strategies. It first introduces the classic monolithic architecture and highlights its core strengths and weaknesses. Later, a popular modern architecture style named microservice-based architecture is explained. Finally, it briefly enlists the core incentives of using software containers with microservice-based architecture.

3.1 Monolithic architecture

Monolithic architecture is the traditional style of building a software system. This architecture has been the core design principle of many software systems currently in use in various businesses and industries. Monolithic architecture intends to compose all software components in one large piece [39]. Monolithic softwares often lack modularity between their different components as they are interconnected and have strong dependencies on one another. These components have obscure boundaries of modularity. Eventually, all of these components or vaguely separated modules are deployed together as a single process.

Generally, a monolithic application may contain many components or modules designed to support different functionalities. The components having similar or related functionalities are usually placed together under logical groups called software layers. For example, the user interface (UI) layer or often called presentation layer, logically groups all the modules related to the UI for a monolithic application. Figure 3.1 presents a general layered architecture of a monolithic application, where different components or modules are logically grouped under software layers. Eventually, all of these software

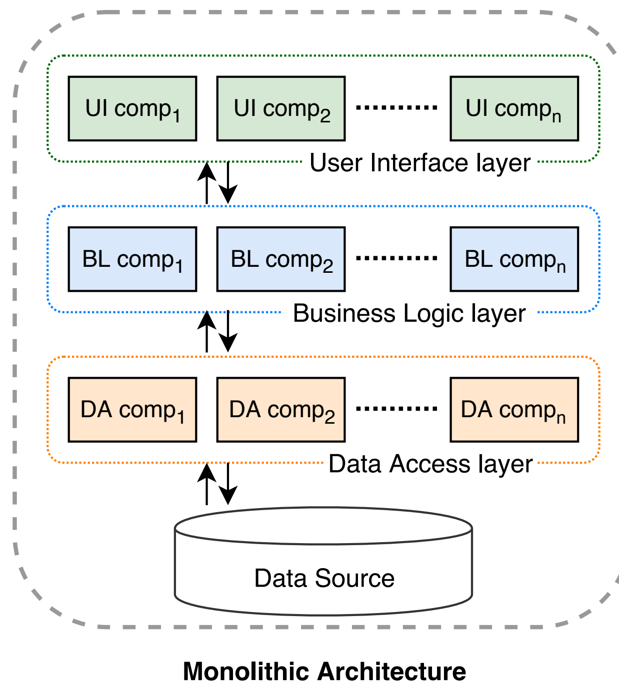

Figure 3.1: Typical layered architecture of monolithic applications

layers are packaged in an autonomous unit for deployment and delivery.

Pros

First, we will analyze some of the advantages of monolithic architecture.

The deployment of monolithic applications is relatively easy as the whole codebase is packaged into a single unit, requiring the same single deployment cycle every time. Similarly, monolithic softwares are easier to develop, test and maintain as long as the codebase is relatively small. Many existing integrated development environments (IDE) are mainly designed to ease the development processes of the monolithic applications [43].

In monolithic architecture, various software components are strongly interconnected. This unification of components minimizes the communication cost between various components and often yields high throughput [39].

Furthermore, small monolithic applications can be scaled up easily using either vertical or horizontal scaling techniques. Horizontal scaling is achieved by running multiple copies of the same monolithic application on multiple servers. Typically, these are placed behind a load balancer, which is a dedicated device with the core responsibility of coordinating the communication between these application servers and the outside world. The load balancer

receives the incoming requests, distributes the requests among the application servers, with the aim to balance the traffic load and finally returns the response/data back to the initiator of requests. On the other hand, vertical scaling involves migrating the application to a machine which has relatively higher or more powerful resources, such as computing power or memory. However, this results into a service down time [44]. This service down time can be reduced by adding a load balancer and hot swapping[1] the server with the new more powerful server.

Cons

The advantages of monolithic applications diminish as they grow in size. A bigger codebase often results into stronger dependencies between its different modules. The strong dependency between various components is called high coupling, which makes the larger codebase more complex and hence, difficult to develop and debug. For example, an inappropriate bug fix in the complex codebase can cause ripple effects and create more bugs, which can be further hard to find and fix. Overall, complex codebase not only cause the developer's productivity to decline but maintaining the code quality also becomes increasingly challenging.

In many software applications, various components have different needs for computational resources at any given time. In other words, different components can have different scalability requirements within the same application. This specific situation is fundamentally undermined in monolithic architecture. All components are treated the same and are thought to have the same scalability criteria. As a result, all components are scaled up or down at the same time, without considering whether a certain component actually requires additional computational resources on not. If any component requires more resources, the whole application must be scaled up.

Similarly, all the components in a monolithic application have the same life cycle. All modules need to be compiled and deployed together at the same time. This single unit design means that a bug in any module or component will break the whole monolithic application. Consequently, the whole monolithic application needs a new deployment, irrespective of the size and relevant module of the bug. Furthermore, if a monolithic application is benefiting from horizontal scaling (replication of application), all instances of the application will be redeployed, once a bug has been identified in any of the components.

In addition to high coupling between components, monolithic applica-

[1] replacing or adding components while the system is still in use [38]

tions introduce high coupling with the technology stack as well. For instance, consider a monolithic application that was initially developed using a specific version of Java Virtual Machine (JVM). Imagine that specific JVM has been succeeded by a newer version. Despite of the fact that using the new version of JVM will have certain performance improvements in the application, the migration to the new version of JVM will not be possible if even a single component is incompatible with the newer version. In short, monolithic applications, often get stuck with the already chosen technology stack and incremental migration to some other technology stack is impossible [42]. Similarly, in some extreme cases, monolithic application might require a complete application rewrite. For example, if the Monolithic application was developed against a platform which becomes obsolete soon [43].

3.2 Microservice-based architecture

Microservice-based architecture aims to build an application as a collection of small independent services, which have their own life cycle and set of available resources. These small services are called microservices and communicate with one another by lightweight communication mechanisms, such as Application Programming Interface (API) calls over HTTP [41]. Microservices are deployed independently in a distributed system.

Microservices introduce clear boundaries of modularity and help developers to focus on one business context at a time in a specific service. Different microservices can be developed by different teams and can have different release cycles. One team can deploy their updated microservice without affecting the rest of the microservices running in the distributed application. Ultimately, maintaining and debugging a particular microservice becomes relatively easy.

The size of a microservice usually depends on the situation and can vary depending on the company and the needs. However, as a general principle, one microservice should be small enough to be handled by one team [56]. Nonetheless, the microservices should not be too small because they introduce additional inter-service communication. This communication can dent the overall performance if there are too many service calls [40]. The execution time of each call will add up to the overall response time of the application. This may lead to a situation where developers have to reconsider the modular boundaries or the scope of the microservices.

Another guideline for identifying the size and scope of a particular microservice is to ensure that the microservices are developed in accordance with the design principle called Single Responsibility Principle (SRP). The

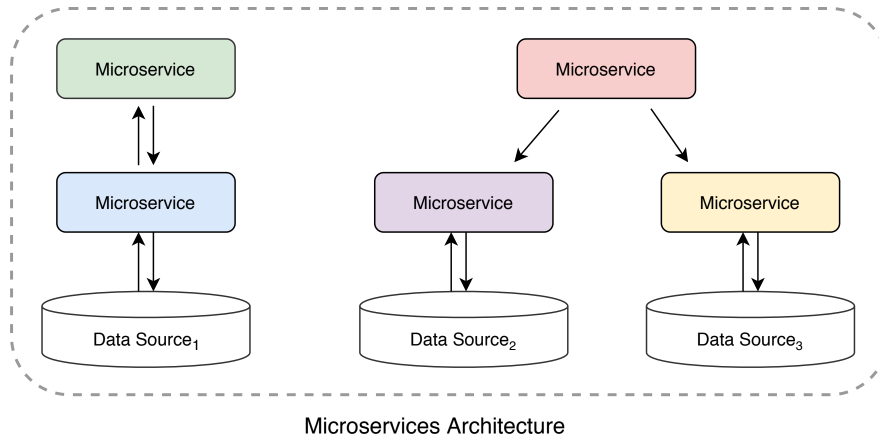

Figure 3.2: High level view of microservices based application

SRP implies that a specific module should have a unique responsibility. The compliance with SRP introduces high cohesion and low coupling between difference microservices. Cohesion indicates the "degree to which the elements inside a module belong together" [60]. On the other hand, the coupling between different software modules indicates their dependency on one another; high coupling means that the modules are highly dependent on one another [1]. A microservice interacts with the second microservice via its exposed interface and does not know the internal details of the second microservice. Hence, microservices are loosely coupled and highly cohesive.

However, the low coupling and high cohesion comes at a price. Consider a certain piece of code which is being used by multiple modules, while creating microservices, the developer either has to create a new microservice for this specific code or duplicate the code in multiple microservices. The first approach increases the inter-service communication, while the second approach goes against another software design principle called Do not Repeat Yourself (DRY). DRY discourages the repetition of the code and emphasises on the code reuse. The repetition of the code has a clear disadvantage of maintaining multiple copies of the same chunk of code, which essentially means that a single change must be replicated on all copies.

Microservice-based architecture goes well with scalability, especially in applications where different modules have different requirements. Microservices enable distributed systems to have fine-grained and flexible scaling. Developer has full control to decide which microservice should be scaled to what extent and using which scaling dimension, i.e., horizontal or vertical. For example, imagine three microservices in a random distributed system, which have different scalability requirements. Suppose one microservice is

getting increasingly high traffic/customer traction and needs rapid scaling. It can scaled horizontally by spinning up more instances of the particular microservice. The second microservice needs more computational power. Thus, it can be scaled vertically by migrating the microservice to a more powerful/resourceful server. Finally, the third service does not require any scaling and efficiently runs on the current resources. Consequently, unlike monolithic architecture, microservices can serve the different scalability needs.

Microservice-based architecture is a better choice for canary deployments [54]. Canary deployments means developers can release specific versions of the software to specific groups of users, mainly to test the features and collect feedback. Microservices enable incremental testing of various features in production environment on actual users. For example, multiple versions of a specific microservice can be deployed to the production at the same time. The team can collect vital user feedback on multiple versions of the microservice from real users. Canary deployments are also possible in monolithic architecture with limited flexibility. In monolithic application, every version of a service, independent of the update size, requires the entire application to be redeployed.

Another distinctive feature of microservice-based architecture is its ability to support multiple technology stacks within the same application. Various microservices within the same application can be developed using different technology stacks or dependencies. For example, two microservices using different versions of Java can coexist in the same microservice application. Microservice-based architecture gives development teams the flexibility to choose the technology stack which is the best in the business context of the microservice. For example, machine learning team can freely choose the Python or the R programming language and does not have to worry about the fact that the rest of the microservices are developed in some other technology such as JAVA or Scala. This can boost the performance of the individual microservice as different microservices can benefit from different languages or technology stacks according to their business context [56].

Similarly, one microservice team can decide to use a database that is most suited for their business need. As shown in Figure 3.2, multiple microservices can have their own databases targeting different database engine types. For example, if other microservices are using a Relational Database Management System (RDBMS), but that is not the optimal choice for a particular microservice, the team can choose to use any NoSQL database such as MongoDB. The clear boundary of modularity and communication over standard protocols like HTTP, allow different microservices having different technology stack and database choices, to coexist in the same application.

3.2.1 Software containers in microservice-based architecture

Containers are best suited for microservices applications [57]. Each microservice can run as a container and use minimum resources. Containers provide isolation between different microservices unless developers explicitly connect them. Additionally, multiple instances of a same microservice container can be run in parallel either on the same host or a different machine. Each microservice running in a container is independent of other microservices. This allows microservices to be deployed in a data center with the same ease and consistency as if they were deployed on a developer's personal computer [10].

In a typical microservice-based architecture, following are some of the distinctive benefits of software containers.

Efficient resource usage

Unlike virtual machines, containers do not have to run the entire operating system. Hence, their consumption of resources is low. You can run more instances of a container on a single physical machine than the number of virtual machines.

Faster start time

The startup time of a container application plays a significant role is building, testing and scaling applications [20]. A container usually starts much quicker than a virtual machine [34]. In microservice application, there can up-to dozens or hundreds of small services, starting all of them in virtual machines will consume a considerable amount of time and resources. Furthermore, the building and deployment of docker containers can be automated, which further speeds up the process and reduces the effort.

Quicker scaling

Considering the small size and footprint of a container, it can be created and destroyed rather quickly when compared to virtual machines. Consequently, if a microservice application needs to scale rapidly, creating more replicas of the required containers would take relatively less time. Similarly, if any of the containers fails for some reason, replacing it with a new healthy replica would be faster than replacing a failed virtual machine with a new one. Thus, containers can reduce the downtime in case a failure occurs.

Portability

The absence of complete operating system, allows a docker container to be ported to a different machine in a different environment easily.

Support

Docker is available for most of the famous operating systems, i.e., Windows, Mac, and Debian etc.

Chapter 4

Container Orchestration

This chapter aims to define the container orchestration and highlights its main features. Additionally, it provides a detailed discussion on Kubernetes, the prevalent orchestration frameworks. Furthermore, it presents a brief overview of another orchestration frameworks named Amazon Elastic Container Service (ECS). This information is important to understand the technical implementation of this thesis.

4.1 What is container orchestration?

As discussed in the previous chapter, modern applications that follow microservice architecture, are composed of several loosely coupled components. These components are usually built as a container. These containers need to work together to achieve the desired functionality. As a result, the efficient management of these containers becomes an uphill task. Furthermore, in large distributed applications, the number of these containers can increase rapidly as the application grows bigger or move towards higher levels of modularity. Consequently, the container management becomes an increasingly difficult challenge. Although, one can apply different automation techniques to reduce the effort of container management, yet these techniques would require writing complex automation scripts. These scripts tend to become progressively difficult to maintain and end up consuming more and more effort in the long run.

Container orchestration is essentially the process of managing the life cycle of the containers, particularly in large and complex environments. Container orchestration allows the administrators to control and automate many of the processes that would otherwise consume a lot of time and effort. Container orchestration tools allow the users to describe the configurations that

the containers and other cluster components should adhere to. The configurations are typically provided in a YAML or JSON file and specify the way the containers should be built, the storage they should be allowed to use and how the communication should be established between containers and different cluster components. Once the configuration is defined by the administrator, the container orchestration framework makes sure that all components match to the specified configuration [22].

Some of the main features of the prominent orchestration frameworks are discussed briefly below [45]:

- **Deployment and provisioning.** When new containers are built with new deployment, the orchestration framework will find the appropriate host for these containers according to CPU, memory and network availability. Once the containers are hosted, the framework will manage the container life cycle. The framework can monitor the health of the host and report any possible fault.

- **Replication and load balancing.** Orchestration framework can create multiple copies of the specified containers and distribute the traffic across all replicas.

- **Availability and fault tolerance.** If the user wants a particular container to be available at all times, the framework can ensure that even if a particular container encounters a failure, a new copy replaces the failed container.

- **Scaling.** Orchestration framework can respond to incoming traffic or workload by automatically scaling the service up or down.

- **Communication.** Orchestration framework can not only allow different containers to communicate over the predefined interfaces but expose the required services out side of cluster as well.

Next, we discuss two of the most common orchestration frameworks used all over the world.

4.2 Kubernetes

The name kubernetes has its roots in Greek language and translates to pilot or helmsman [32]. Kubernetes is a platform for container orchestration, which was originally developed by Google. Kubernetes provides effective deployment and management of container's life cycle in clusters. It emerged

from a tool called Borg [59], which had been used for cluster management by Google for several years. In 2014, kubernetes was released under an open source licence. Currently, Kubernetes is extensively used world wide and has a large and growing ecosystem, which means that the essential Kubernetes support and tools are readily available [32]. Kubernetes provides services that are helpful to achieve predictability, scalability, and availability.

Kubernetes allows you to define declarative configurations that describe the way containers should run and interact with other containers or applications. It can automatically scale your application up or down as per the requirements. Kubernetes allows you to manage the traffic between different containers, which is helpful in many scenarios. For example, when you want to rollback a faulty deployment or you want to release different versions of you application for testing and gathering feedback. Kubernetes, not only brings ease in deployment, scaling, and maintaining the applications but introduces portability as well. For instance, if a containerized application is hosted in on-premises, Kubernetes can allow an efficient migration to a cloud provider environment.

Kubernetes can be conceptually seen as system having layers in which each higher layer abstracts the complexity of lower layer [25]. Kubernetes groups together the physical and virtual resources in the cluster and manages their communication. As per the configurations provided, the machines in the cluster are given a role that they need to perform. These roles and some of the other kubernetes concepts are explained below:

4.2.1 Master

One server is assigned the role of master. This server acts as the brain of the cluster and performs most of the containerization logic [25]. In an environment where availability requirements are high, the master role can be given to a small group of servers too. The application containers are usually run on different servers than the master server, which ensures a simple and easy cluster management. Master server, being the primary contact point of administration, has many components that collectively manage the cluster. The responsibility of these components range from receiving the end user requests to monitor the health checks of other cluster resources. The components can either be placed solely on a particular server or one can choose to deploy different components to different servers. Some of the main components of master server are briefly described below.

etcd is a configuration store that is available globally. It is a distributed key-values store which saves the configuration data. This can be used by

other nodes for service discovery and staying up-to-date with the latest configurations. Additionally, the store tracks the cluster state and provides functionalities like distributed locking and leader election. This component is usually placed on a single machine but can be distributed across multiple servers provided it can be accessed by other nodes [25].

kube-apiserver being the main management point, is one of the most important components of the master server. It receives the data on REST interface and updates the relevant values in cluster stores such as etcd. API server enables the communication between master and other cluster components and helps maintain the cluster health [53]. This API server is also the communication bridge between the administrators and the Kubernetes. The administrator can use tools like kubectl from their local computer to communicate and manage the cluster resources.

kube-scheduler is the main process designated to assign workload to different nodes. The scheduler also keeps track of the total capacity of nodes. Every time a new workload requires placement, this scheduler examines the available capacity of existing nodes and assigns the workload to the most suited node(s). The selection of node(s) is affected by different software, hardware, and network constraints. The scheduler aims for efficient load distribution so that the workload does not exceed the available resources.

cloud-controller-manager One of the distinctive features of Kubernetes is its ability to be deployed in different environments using different infrastructure providers. *cloud-controller-manager* essentially provides a way for the Kubernetes to interact with resources provided by heterogeneous cloud providers by running cloud-specific control-loops [29]. This enables Kubernetes to maintain a consistent state, manage the existing cloud resources, and create more cloud services when needed [25].

4.2.2 Pods

A pod corresponds to one or more containers that should be controlled as a single entity within the application. It is the fundamental building block in Kubernetes object model. All containers within a single pod will always be created, terminated and scheduled together. In other words, these containers will not only have the same life cycle but will share the same storage and network resources too[1]. The containers inside a pod will use localhost for

[1] `https://kubernetes.io/docs/concepts/workloads/pods/pod/`

internal communication and use the shared network resources like IP address and port number for external communication.

Conceptually, the pod can be seen as a single monolithic application, even if it is hosting multiple containers [25]. Consequently, a failure in any of the containers will render the whole pod as faulty. Thus, Kubernetes will replace the complete pod with a healthy replacement. Similarly, scaling in Kubernetes always results into creating/deleting the pods.

4.2.3 Node Server

The servers that host the pods are called node servers or nodes [2]. Nodes are controlled by master server(s). Their primary responsibility is to provide the run time environment for the pods which they are hosting. Following are the node server components which are critical for network configuration and establishing the communication between master and node servers.

Container Runtime is required to run the containers and execute the actual workload. Normally container runtime is provided by docker. However, Kubernetes provides the support for other container runtimes like CRI-O, Containerd, and Frakti [26].

kubelet is the service responsible for executing the containers. It communicates with control plane services including etcd store. This service receives the workload from master components. Kubelet reads the PodSpecs which typically is in the JSON or YAML file. It creates the pod according to the specifications and controls/monitors the runtime for the container. It can launch or destroy the containers if required.

kube-proxy service resides on each server node and is responsible for host subnetting and ensuring the availability of services to other components. This component provides primitive load balancing solution with low performance overhead. Working on transport layer, kube-proxy can perform simple and round-robin TCP, UDP, and SCTP stream forwarding among different backends [28].

Figure 4.1 illustrates the architecture of Kubernetes. Master server is hosting components like API server, scheduler, manager controller and etcd store. Whereas, the node servers are hosting pods along with node components.Each node is running its own designated containers.

[2] `https://kubernetes.io/docs/concepts/architecture/nodes/`

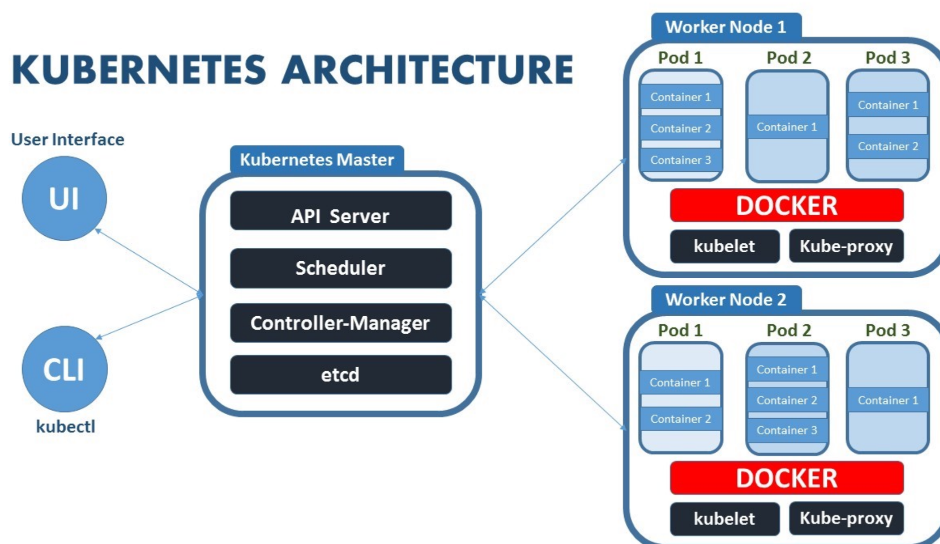

Figure 4.1: Overview of the various components in Kubernetes architecture [33]

4.2.4 Replication Controller

High scalability and availability often means running multiple copies of a single pod. Every pod is created from a pod template which contains a set of rules for the pod. One pod template can be used to create many identical pods. These identical pods are called replicas and are managed by replication controller to make sure that the required number of replicas are running at any given time. It can scale up the pods by running more replicas of the pod, or scale down by removing unnecessary pods.

Additionally, replication controller helps in fault tolerance. It has the capability to detect the pod failure and it can create a new identical pod and replace it with the faulty pod [30]. Consequently, using a replication controller is beneficial even if the application needs to run only a single pod. The life cycle of replication controller is independent of the pods it monitors, meaning the replication controller can be terminated with/without terminating the relevant pods.

4.2.5 Deployment

A deployment is the recommended way to deploy on Kubernetes. A deployment updates the pods to reach the desired state at a controlled rate. A deployment can either create new replicas or utilize the resources of the old

deployments[3].

4.2.6 Services

A Kubernetes service is a component that groups the pods (performing the same function) into a single logical entity. The service defines the policies as how a certain set of pods should be accessed. The service keeps track of these pods and routes the traffic to them. Unlike the pods, the service has a stable endpoint which is exposed to the consumers. Consumers do not communicate directly to the pods, because a particular pod can be removed or new pods can be added during the run time. Any deployment can add, remove or update these pods. Thus, the consumers communicate to the service via RESTful operations. The service then forwards the requests to the relevant pod(s). This abstraction allows the services to do internal load balancing and yields a better discover-ability for the consumers.

4.2.7 Ingress

Ingress is responsible to manage the external access to the cluster services. Ingress assigns an externally-accessible URL to the service, and performs load balancing. It accepts outside requests on HTTP or HTTPS endpoints and forwards the requests to the relevant services in the cluster. One can define the rules about how the traffic should be routed[4].

4.2.8 Web User Interface

Kubernetes has a web-based user interface also called the dashboard. This interface can be used by the administrators to manage the deployments and other cluster resources. It also allows the users to troubleshoot the possible problems and take corrective actions.

4.3 Elastic Container Service (ECS)

Elastic container service is a container management service provided by Amazon, which allows a container-based application to run in a cluster. This cluster can be hosted in either Elastic Compute Cloud (EC2) or in a serverless environment such as Amazon Fargate. ECS enables an easy and efficient

[3] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[4] https://kubernetes.io/docs/concepts/services-networking/ingress/

deployment of the containers-based application. It provides a centralized service, which continuously manages the state of the cluster. ECS entails many other features, such as load balancing, auto scaling, implementing isolation policies and ensuring availability. ECS provides an excellent platform for deploying and managing the sophisticated applications built on microservices architecture [5]. Some of the key concepts of ECS are explained briefly below:

4.3.1 Task Definition

The first step to run the container based application on ECS, is to create a task definition. Task definition is a JSON file that contains the configurations required to run the containers. One task definition can describe at maximum ten application containers [4]. The configurations include specification of the base docker image and data volumes, CPU and memory allocation, mapping of ports, and setting up the environment variables for the specified container. Additionally, task definition specifies the launch type. The launch type determines whether the containers should be run on EC2 or serverless environment like Fargate. The specification of configuration parameters can vary based on the specified launch type.

4.3.2 Task

Instantiation of a task definition within a cluster yields a task. One task definition can be used to create as many tasks as required. Tasks running in serverless environment, run is isolation with sharing CPU, memory or network interfaces with other tasks.

4.3.3 Scheduler

The scheduler is the component that takes care of the placement of the tasks in the cluster. For example, if the launch type is EC2, the scheduler decides which task should be run on which EC2 instance. ECS also allows you to define the scalability and availability constraints that should be considered while scheduling the tasks [58].

4.3.4 Cluster

A cluster represents a logical grouping of resources. All tasks are run inside the clusters. The responsibility to manage the cluster resources depends on the launch type. For example, if the environment is serverless, ECS will take care of the management of the resources. However, with EC2 launch

type, the cluster comprises of the ECS container instances which should be managed by the administrator.

4.3.5 ECS container instance

The term ECS container instance is used for an EC2 instance that is running the ECS container agent. Each container instance is created from an already specified container image, which is fetched from a predefined container registry.

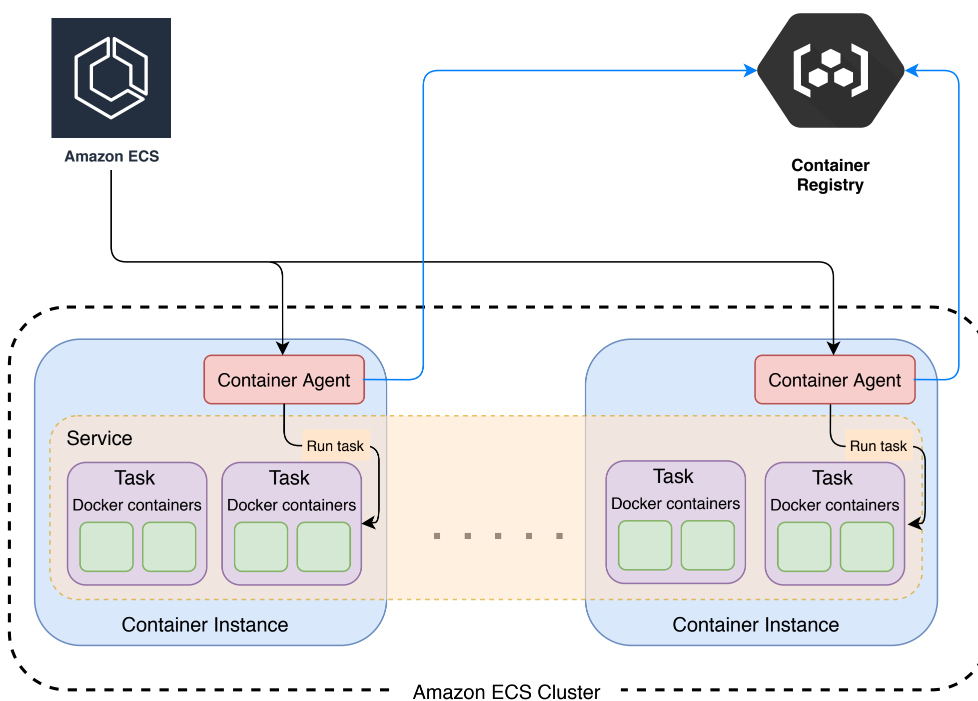

Figure 4.2: Overview of Amazon Elastic Cloud Service

4.3.6 Container Agent

The container agent is responsible for the communication between the cluster resources and the ECS. The container agent runs on each resource and sends the status information back to the ECS. In case of EC2 launch type, the information about running tasks and resource utilization is also communicated to ECS by the container agent. It can start and stop a particular task when requested by ECS.

4.3.7 Service

A service is deployed on the cluster and allows you to manage the instances of a task definition. For example, you can define the maximum or minimum number of instances that should be running in parallel. The service ensures the availability by maintaining the specific minimum number of instances. It responds to the failure of any instance and replaces the failed instance with a new one. Additionally, a cluster can run multiple services, which means If you have multiple applications that you want to deploy and you want to optimize the resource utilization. You can choose to run them within the same cluster.

As shown in Figure 4.2, the ECS communicates with the container agent which is running on all instances. The container agent downloads the container image and runs the tasks when instructed by ECS. On the other hand, the service manages all instances running inside a cluster.

Some of the major advantages of ECS are listed below [24]:

- **Easy cluster management.** ECS provides easy management of clusters which contain docker containers. It maintains the information about the cluster state can scale clusters across multiple availability zones.

- **Flexible Scheduling.** The ECS scheduler ensures a balanced availability and utilization. Additionally, ECS allows you to use any existing open source scheduler as well.

- **Resource Efficiency.** ECS aims for the efficient resources utilization. One can run various unrelated containers on a single EC2 instance. these containers can coexist while using the resources of the same underlying EC2 instance.

- **Resource Efficiency.** ECS allows the usage of many other Amazon Web Services (AWS) resources like Virtual Private Cloud (VPC), resource tags, and Elastic IP addresses etc.

- **Security.** The EC2 instances run in Amazon VPC and benefit from other AWS security features, such as IAM role and security groups.

Chapter 5

Invoice-based B2B payments

This chapter introduces business-to-business (B2B) payments. It provides a high level discussion on an invoice-based B2B payment service, which is used as the case study for this thesis. It describes the main entities that are typically involved in such a service. Additionally, it presents a brief overview of the life cycle of an invoice-based B2B purchase. This discussion is important to understand the business domain of the product, which has been migrated to microservice-based architecture in this thesis.

5.1 What is B2B payment?

The term B2B refers to a commercial transaction between businesses. In other words, when both the consumer and the producer of the exchanged goods or services are businesses, the said transaction is termed as a B2B transaction [36]. There are many payment methods which have been traditionally used to transfer the funds from the buyer to the seller. Some of them are highlighted below [50]:

- **Checks** are the traditional payment method, where the buyer gives a check to the seller, who has to deposit it in the bank to transfer the funds from buyer's bank account to the seller. This process is slow and usually takes a few days.
- **Wire transfers** are relatively faster than the checks. The funds are transferred through a financial network in a few hours.
- **Credit/Debit cards** allow the buyer to buy products conveniently and the payment can be completed in one or more billing cycles.

- **Payment gateway** requires the buyer to pay during the checkout process.
- **Invoices.** The seller sends an invoice to the buyer once the goods are delivered to the buyer. The invoice contains the details of the purchase items, total amount, and the payment due date.

All of the above mentioned payment methods are used in various industries and have their own advantages and disadvantages. However, we will mainly focus on the invoices as the payment method for B2B transactions.

5.2 Enterpay's invoice-based B2B payment Service

Enterpay Oy[1] is a Finnish company, which offers a payment service that provides invoicing capability to the sellers for their business customers. We will refer to this service as the *payment service* in the further discussion. The sellers can sell their goods without having to setup a credit policy, invoicing process, reminding process, and collection process. All these processes are automated by the payment service. Additionally, the payment service takes away the credit risk for the seller. All purchases are insured and the seller is guaranteed to receive the payments.

The payment service has been developed over the last 7 years and follows monolithic architecture. Enterpay is growing its business and expanding its services to different countries in Europe. Consequently, the ability to scale rapidly and handle increasing service requests, are currently one of the top priorities. This thesis focuses on migrating the payment service to microservice-based architecture to enable efficient scaling for the growing requests.

In order to understand how the payment service functions, it is important to know the main actors involved in the payment service. These actors are listed below:

- **Merchant** is the seller of the goods or services. The merchant usually has an electronic webshop/website, which is integrated with the service provider, i.e., Enterpay.
- **Buyer** is the company that needs to purchase goods from the merchant.

[1] `https://www.enterpay.fi/`

- **Payment Service Provider (PSP)** provides the merchant a portal which integrates different payment methods or services. Some merchants do not integrate directly to the payment service, rather they are integrated to a PSP, which already has an integration with the payment service.
- **Financing partner** is usually a bank or a financial service company. Financing partners are customers of Enterpay and are responsible for taking the credit risk and collecting the payment from buyers.

5.2.1 Life cycle of a purchase

As a prerequisite of using the payment service, the merchant needs to integrate to the *Purchase API*, which is a REST API exposed by Enterpay. Once the integration is done, the merchant will enable Enterpay as one of the available payment methods on its webshop.

Figure 5.1 provides a high level view of the purchase life cycle. The buyer creates a shopping cart on the merchant's webshop. During the checkout stage, the buyer chooses a payment service as his/her preferred payment method. Merchant sends the shopping cart data to the payment service, either directly or via PSP. The payment service validates the data received from the merchant. If the validation checks are passed, the buyer is redirected to the payment service's website.

After redirection, the payment service performs the buyer identification/authentication. The payment service sends a One Time Password (OTP) on buyer's phone number or email address for authentication. After this step, the buyer specifies the invoicing details that are necessary to deliver the invoice correctly. For example, the buyer can choose whether he/she wants the invoice in paper form, over email, or as an electronic invoice[2].

As the last step, the buyer confirms the purchase. During the confirmation, the payment service makes the credit decision whether the buyer should be allowed to make this purchase or not. This credit decision is based on various parameters such as the credit rating of the buying company, allowed credit limits of the buying company, credit limits of the buying person etc. If the credit decision is positive, the buyer is allowed to make the purchase. Finally, the buyer is redirected back to the merchant's webshop, while the payment service informs the merchant about the purchase status.

After a successful purchase, the merchant prepares the goods for delivery. Once the merchant is ready to ship the goods, it can call the *Invoice API* to

[2]An electronic invoice goes directly in the buyer's bank account. This kind of invoice might not be available in all countries.

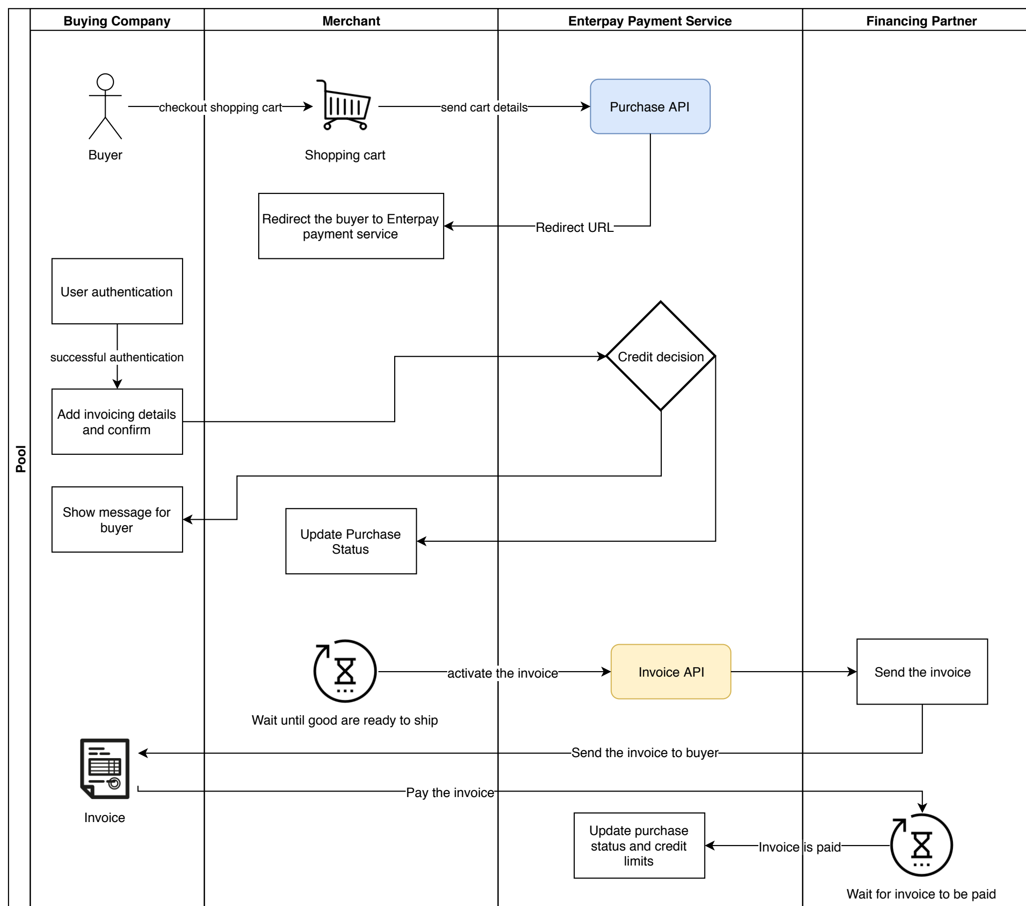

Figure 5.1: Overview of the life cycle of an invoice-based B2B purchase

activate the invoice. The payment service sends the invoicing information to the relevant financing partner, which sends the actual invoice to the buyer. The buyer usually has 14-60 days to pay the invoice. Once the invoice is paid, the financing partner informs the payment service. The payment service then updates the purchase status and the relevant buyer credit limits.

5.2.2 Benefits of invoices for businesses

In many countries such as Finland, businesses can deduct the Value Added Tax (VAT) on their purchases [51]. In simple words, businesses are reimbursed for the VAT paid on their purchases. However, they have to report it while settling their annual tax return. Therefore, it is very important for companies to save the receipts of their purchases. These receipts should have the details such as what was bought and the VAT paid on individual items.

These receipts are collected and the tax information is manually added in the company's tax files.

Unfortunately, many of the traditional payment methods are not favourable in this situation. For example, the Credit/Debit cards have a fundamental drawback that they do not record the details of purchase items. Rather, only the final due amount is displayed on the receipt. Consequently, companies either lose the VAT reimbursement, or they have to ask for a separate receipt from the merchant, which has the purchase item details. In any case, these detailed receipts do not eliminate the cost of adding the tax information manually to the *bookkeeping*[3] system.

On the other hand, the invoice-based payment methods such as Enterpay's payment service provide an efficient solution to this problem. The invoice contains the detailed information about the purchase items and can directly be sent to the buyer's bank account. The electronic invoice can be imported to the buyer's bookkeeping system and can automate the process of adding the relevant VAT taxes.

Another distinctive benefit of invoices, is the efficient management of purchase rights. In big companies, many people have the purchase rights and buy for the company on regular basis. Without invoice-based payments, the company would require to obtain a separate business credit card for these procurists. Similarly, if the employee leaves the company, the issued credit card needs to be cancelled. Consequently, managing these credit cards can be slow and an inconvenient task.

Finally, with invoice-based payments, companies do not have to pay upfront. The purchase is done on the credit and the buyer does not get the invoices until the goods are delivered.

5.2.3 Features of the payment service

Enterpay's payment service has the following additional features which are helpful for the buyer, the merchant, and the financing partners.

- **Purchase reports.** The buyer can login to the payment service and access the list of purchases made by his/her company. In some cases, the buyer can request for a change in the invoicing details of a particular purchase or request a copy of the invoice.
- **Sales reports.** The merchant can also login to the payment service and see the sales report. The merchant can make cancellations/refunds

[3]bookkeeping is the process of maintaining or recording all of the financial transactions such as sales and purchases.

of the purchases.

- **User management.** Both, the buyer and the merchant can register more users to the payment service. Additionally, payment service allows them to define various privileges and purchase limits on the users of their organisation.

- **In-Store Mode.** Enterpay provides a tool to create purchases in the payment system. This is specially beneficial for the merchants which do not have a webshop but want to sell their goods using invoices.

Chapter 6

Implementation

This chapter provides a detailed discussion of the technical implementation for this thesis. It explains the main migration stages of Enterpay's payment service from monolithic to the microservice-based architecture. Additionally, it briefly describes the tools and commands that were used in these stages.

6.1 Migration to microservice-based architecture

The migration from monolithic to microservice-based architecture was primarily done in three stages. Each stage involved introducing new tools to the technology stack of the payment service. In the following discussion we describe each of the stage in detail. Additionally, we explain some of the challenges faced and how they were overcome.

6.1.1 Introducing Docker

The first step of the migration process was to introduce Docker in the technology stack of the payment service. This step was primarily performed on the developer's machine. We installed the Docker by downloading and installing *Docker Desktop for Mac* from the official Docker website [13].

The deployment of the current monolithic application begins by building a Java ARchive (JAR) file, which includes all of the classes and dependent libraries for the monolithic application. This file is then deployed as a service on a Linux machine. Introducing Docker in the current deployment model, meant that we encapsulate the same JAR file inside a Docker image. Consequently, we created a *.dockerfile* as shown in File 1. The dockerfile includes

the information on the base image, required libraries for the runtime, and the application configurations required by the JAR file.

```
FROM <base_image>

WORKDIR /workspace/
RUN apt-get -y update
RUN apt-get install -y mysql-client
RUN apt-get -y install openjdk-8-jdk
COPY ./project ./project
COPY ./conf ./conf
COPY ./ui ./ui
COPY ./run_jar.sh ./<fat_jar_file.jar> ./
EXPOSE <port_no>
ENTRYPOINT ["./run_jar.sh"]
```

File 1: .dockerfile for the monolithic application.

Once the *.dockerfile* was ready, we built the Docker image. The following command creates a Docker image and assigns it the tag *latest*. At this stage the Docker image was saved locally on the developer's machine.

```
#builds the docker image
$ docker build -t enterpay-app ./
```

This image was used to create a container. This following command ran the whole monolithic application inside a single Docker container.

```
#runs the docker image on localhost:<port_no>
$ docker run -p <port_no>:<port_no> enterpay-app
```

6.1.2 Modularization - Transition to Microservices

In the second stage, we identified some of the modules that should be extracted and run independently as microservices. The choice of these modules was based on parameters such as, scalability needs, visibility of the module to the end user, and effort required to extract and decouple the modules. For the scope of this thesis, we divided the monolithic application into the following microservices:

- **User Interface (UI).** This service is responsible to serve the HTML content.
- **ConnectIn.** This is a REST API which is in charge of receiving the purchase requests from merchants (machine-to-machine communication).

- **Background-actors.** This service sends invoicing data to the financing partners. It has a specific requirement that only one instance of this service should be running at any given time. In simple words, this service should be available at all times but must never be scaled to multiple instances.

- **Main server.** This is the part which has the rest of the modules of the monolithic application. In future, we will split this further into more microservices.

- **Merchant API.** This is a REST API which is exposed to merchants to perform operations like updating invoice details, refunds, and cancellations.

Each module was built as a stand-alone independent application, which presented the following two main challenges:

- **Inter-Module communication.** Before extraction, one module could conveniently import the assembly references of other modules to consume the offered functionality. Although these assembly references introduce build dependencies, this arrangement is easy and efficient as all of these modules are built under the same monolithic application. However, after the extraction, the build dependencies are not desirable. Consequently, we created additional REST interfaces for these independent applications to establish communication and collaborate on a given task.

- **Base modules.** Some modules of the payment service serve as the base for application structure and are used by other modules. For example, a module named *Common Module* contains different helper classes that are used in different other modules. One such helper class is named *LogUtils*, which contains the logic to write a given entry in the *log* table. In the monolithic application model, the common/base modules are used as assembly reference by other modules. Hence, there is only one copy of the base module. However, after extraction, this base module needs to be available to all of the independent applications. Extracting this module as a separate microservice will introduce a considerable communication between different microservices, which is not desirable. Thus, we allowed the replication of this base module in the extracted modules.

We created a separate dockerfile for each independent application, to treat it as a separate microservice. The newly introduced REST interface are kept

private and are only visible to other microservices. The details about the intermediate implementation steps are list under appendix A.

6.1.3 Deploying to Kubernetes cluster

Once the monolithic application is converted to a microservice-based application, we deployed these microservices to a Kubernetes cluster on Amazon Web Services (AWS). The goal of introducing Kubernetes was to enable efficient deployment and orchestration of these microservices. The choice of AWS as the deployment platform is governed by the fact that AWS is already a part of the current technology stack of Enterpay's payment service.

We created the deployment and service resources for each microservice. Later, we deployed these resources on the Kubernetes cluster. File 4 presents an example of the deployment and service resource for the *Main Service*. Additionally, we used the *Web UI* for Kubernetes, which is a convenient interface for deploying/managing the Kubernetes services and resources. For example, this tool can be used to create, modify or scale the deployments[1].

The Docker images of the microservices were stored in a cloud registry. The images are saved under private repositories and can only be accessed through valid credentials. Kubernetes allows the users to create a *secret* object, which is used to store passwords, Secure Shell (SSH) Keys, and other authentication tokens [31]. Consequently, we created a secret object on Kubernetes cluster named *regcred*, using the following command. This regcred object is specified under the Kubernetes deployment resources, and is required to pull Docker images from private repositories.

```
 # creates the regcred secret on Kubernetes using
 # the docker credentials
$ kubectl create secret generic regcred \
    --from-file=.dockerconfigjson=<path_of_docker-config.json_file> \
    --type=kubernetes.io/dockerconfigjson
```

Furthermore, we created an *ingress* resources as shown in the File 2. The ingress resource implements the API gateway pattern and acts as a single entry point for all of the end user requests. It routes the request to the relevant services that are not visible directly to the end user [9].

[1] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

```
kind: Service
apiVersion: v1
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "*"
    service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout: "120"
spec:
  externalTrafficPolicy: Local
  type: LoadBalancer
  selector:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
  ports:
    - name: http
      port: 80
      targetPort: http
    - name: https
      port: 443
      targetPort: https
---------------
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress
spec:
  rules:
  - host: <public_url_of_payment_service>
  - http:
      paths:
        - path: /connectin
          backend:
            serviceName: connectin-service
            servicePort: <connectin_service_port>
        - path: /
          backend:
            serviceName: main-service
            servicePort: <main_service_port>
```

File 2: ingress.yaml: configuration for the ingress controller and the ingress resource

Figure 6.1 elaborates the high level architecture of the payment service after moving to microservice-based architecture. Different microservices are running independently and have their own replication policies. For example, the Main service is running three replicas and the Actor Service is running only one. The deployment resources for these microservices control the restart policy, which is set to ‘Always’. This means that in case a pod fails for some reason, the deployment will create a new pod automatically to replace the failed one. All services in Kubernetes cluster are by default, kept

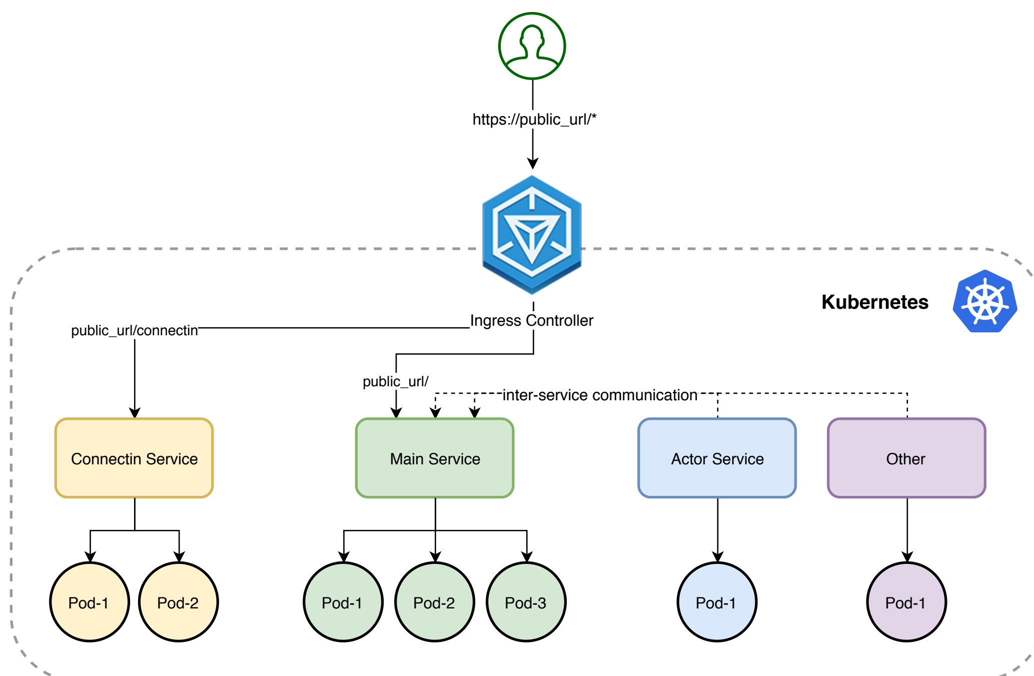

Figure 6.1: Overview of how ingress controller directs the traffic to different services

invisible to the end user. However, these services can directly communicate with one another, when required.

It is important that in a microservice-based architecture, only the relevant services are exposed to the end user. Furthermore, these service should be exposed on a specific public Uniform Resource Locator (URL). Therefore, the ingress controller receives all customer requests on a specific base URL. It parses the URL and identifies the relevant service which should handle the particular request. Ingress controller acts as a load balancer and forwards these requests to the relevant service objects. Additionally, ingress controller is used to manage the visibility of the microservices to the outside world. For example, as shown in Figure 6.1, ingress controller only exposes the *Main* and *Connectin* services to the end user. The rest of microservices continue to run privately and collaborate with other microservices on a given task.

Chapter 7

Evaluation

This chapter presents the evaluation of the microservice-based architecture developed in this thesis. We first provide the details of the experimental setup and methodology. Then explain the experiments conducted as well as the reported results.

7.1 Experimental setup and methodology

To compare the monolithic and microservice-based architectures, we setup an environment where we can run our experiments. We aimed to create a setup where both architectures have the same physical resources, so that the results are not biased.

- **Microservice-based architecture.** A kubernetes cluster was created on Amazon Web Services (AWS), with exactly one master and two worker nodes. The instance type of master node was *m3.medium* and the worker nodes had *t2.small*[1] as their instance type. Both of the worker nodes were configured to save the data in a single instance of MySql based Relation Database Service (RDS). The instance type of the RDS machine was *db.t2.small*[2]. The motivation behind choosing these instances types was to create an experimental setup, which resembles the actual test and demo environments of the payment service.
- **Monolithic architecture.** We created an *Auto Scaling group*, which consists of Elastic Compute Cloud (EC2) instances. These instances can be managed and scaled automatically. An Auto Scaling group allows to configure the scaling policies based on different parameters

[1]Specs: `https://aws.amazon.com/ec2/instance-types/`
[2]Specs: `https://aws.amazon.com/rds/instance-types/`

such as CPU and memory usage. Additionally, one can configure the minimum, maximum, and desired number of EC2 instances for the Auto Scaling group [3].

An Auto scaling group requires an Amazon Machine Image (AMI), which is be used to create new EC2 instances for scaling out. Thus, we first deployed our monolithic application on an EC2 instance and created an AMI. This AMI was configured with our Auto Scaling group. Finally, we added an Elastic Load Balancer (ELB), which will distribute the incoming requests among the EC2 instances running inside the Auto Scaling group. Figure 7.1 shows the arrangement of different components of experimental setup.

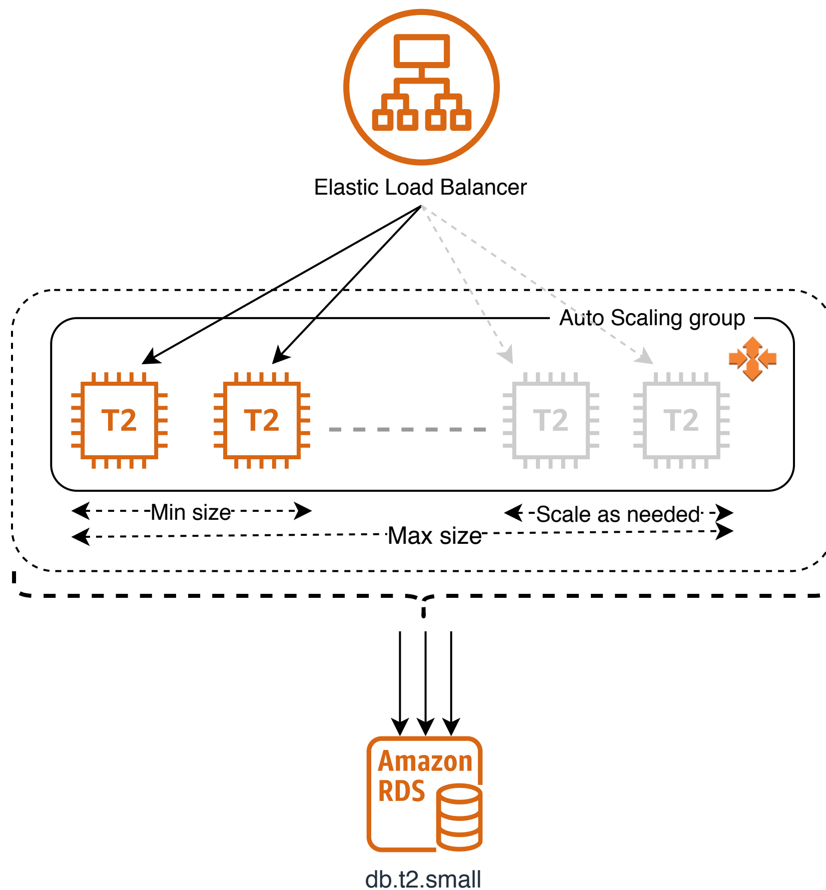

Figure 7.1: Experimental setup for the monolithic architecture

7.1.1 Hypothesis and metrics

We formed the following three hypotheses to evaluate the performance of microservice-based architecture. These hypothesis were tested in the experiments.

- **Hypothesis 1:** the microservice-based architecture should recover from instance failure faster than the monolithic architecture. Accordingly, we decided to measure the average time to recover from the instance failure.
- **Hypothesis 2:** the microservice-based architecture should scale faster than the monolithic architecture. Consequently, we decided to measure the average time to scale out different no of instances in both of these schemes.
- **Hypothesis 3:** even though the microservice-based architecture introduces communication costs between different microservices, the overall response time of service requests should be comparable to the monolithic architecture. As a result, we decided to measure the average response of a specific type of service requests, which involves communication between exactly two microservices.

7.2 Experimental results

7.2.1 Time to recover

In this experiment, we measure the average time to recover from instance failures for both monolithic and the microservice-based setups. The Auto Scaling group is configured to have exactly four instances of the monolithic application. Similarly, the deployment object in the Kubernetes cluster was configured to have exactly four replicas of the microservice.

We used bash scripts to terminate the EC2 instances and pods. The variations of the experiment involved termination of one,two, and three instances. We measured the time since the termination command was issued until the instance was recovered and the application was available to the end user. For each variation of this experiment, the measurements were taken five times and the standard deviation was calculated with the precision of milliseconds.

As shown in Figure 7.2, the average recovery time of microservice-based architecture is significantly smaller than the monolithic architecture.

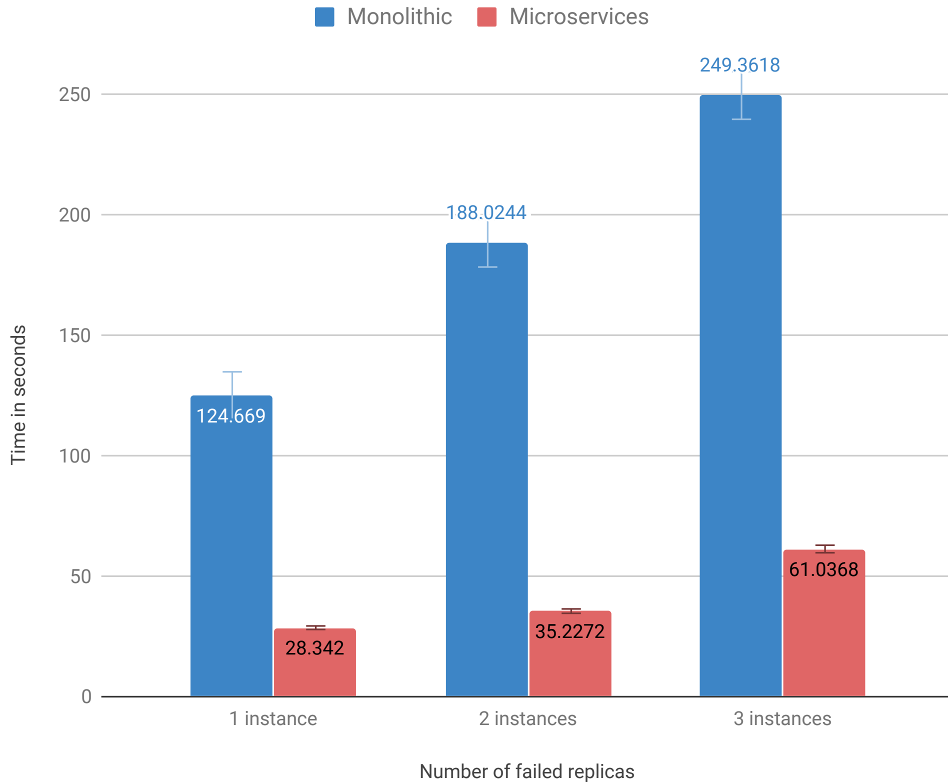

Figure 7.2: Average time to recover from instance failure(s)

7.2.2 Time to scale

In the second experiment, we measured the average time to scale out a given number of instances for both the monolithic and the microservice-based setups. The Auto Scaling group was configured to have exactly one instance of the monolithic application. Similarly, the deployment object in the Kubernetes cluster was configured to have exactly one copy of the microservice.

We used bash scripts to update the configuration of the Auto Scaling group and the Kubernetes deployment object. The variations of the experiment involved concurrently scaling one, three, and five instances. We measured the time since the scaling command was issued until the all of the new instances were available to handle the service requests. For each variation of this experiment, the measurements were taken five times and the standard deviation was calculated with the precision of milliseconds.

As shown in Figure 7.3, the time to scale for the microservice-based ar-

chitecture is smaller than the monolithic architecture. However, we noticed that when more number of replicas are created, the difference of scaling time between both architectures becomes smaller and smaller. The explanation for this behavior is the fact that with Auto Scaling group, scaling involves creating/adding new EC2 instances. In contrast, for the microservice-based architecture, we did not add new worker nodes in the Kubernetes cluster. This essentially means that the new replicas were scaled using the resources of already existing worker nodes.

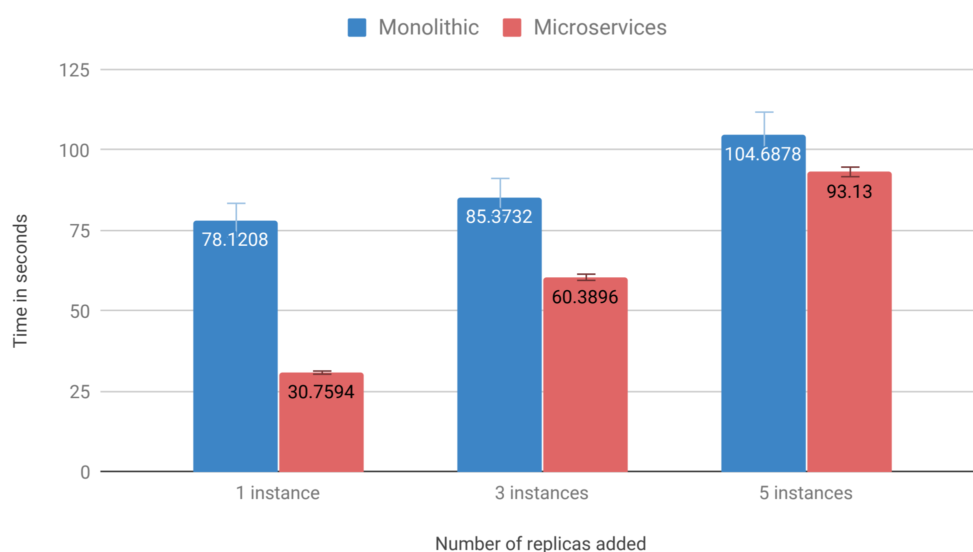

Figure 7.3: Average time to scale out given number of instances

7.2.3 Response time for service requests

In this experiment, we measured the average time for the service requests. We created a service request that created a purchase object in the payment service and redirects the buyer to the payment service where he/she can add invoicing details of the particular purchase. In practice, this service request is initiated by merchants. This request involves processing of approximately 80 string values, saving the values in different tables in database and returning a result containing approximately 20 string values. Furthermore, the processing of this request involves communication between two microservices (Connectin Service and Main Service) over REST APIs.

The Auto Scaling group was configured to have exactly two instances of

the monolithic application. Similarly, the deployment object in the Kubernetes cluster was configured to have exactly two replicas of each microservice.

We used bash scripts to send concurrent service requests. The variations of the experiment involved sending 100, 200, 500, and 1000 concurrent service requests. We measured the time since the script was executed until all of the service requests were processed. For each variation of this experiment, the measurements were taken five times and the standard deviation was calculated with the precision of milliseconds.

As shown in Figure 7.4, the average response time of both architecture is comparable. The communication between the two microservices has resulted in a small delay in the response. We noticed that the response time increases with the increase in the concurrent service requests. The explanation for this behavior is the dependency on the database. The database used for this experiments allows a maximum of 150 connections.

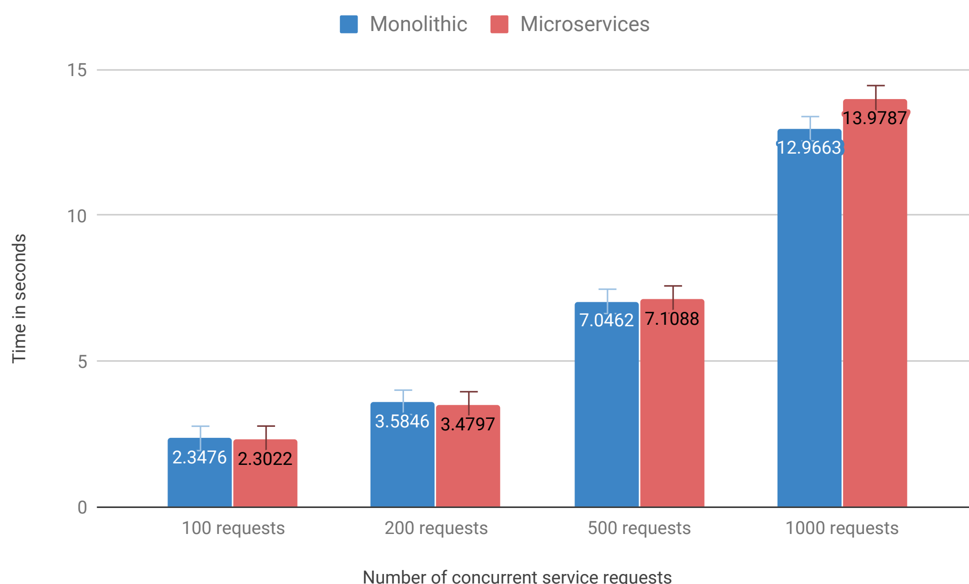

Figure 7.4: Average response time for concurrent service requests

7.3 Considerations on flexibility

The second most prominent challenge faced by Enterpay's payment service is their tight coupling with the current technology stack. The monolithic application was built using Scala, AngularJS, and a specific version of Java. How-

ever, some of the new features require a newer version of Java and software such as Python and ReactJS. With the introduction of microservice-based architecture, these technologies can easily be added in the current technology stack. The design of the microservice-based architecture described in this thesis has allowed the realization[3] of new or improved features for B2B payments.

- **PinCode generation** - This microservice is developed in Java through the Spring framework. It is responsible to create pin codes for authenticating the buyer. It is important to note that this microservice uses a different version of Java than the other Java based microservices running in the payment service.

- **Risk evaluation** - This is a REST API written in Python. The motivation of choosing the Python is the availability of libraries for machine learning algorithms. Prior to microservice-based architecture, this API was deployed to a separate EC2 instance.

The author of this thesis created separate Docker images for these microservices and deployed them in Kubernetes cluster in AWS along with the others. Thus we can conclude that microservice-based architecture is not dependent on the old technology stack and has gained the required flexibility in choice of technologies.

[3]These features were not realized by the author of this thesis. Nevertheless, we leverage them to assess the flexibility of the microservice-based architecture.

Chapter 8

Conclusions

Conventional payment methods such as Checks and Debit/Credit cards are inefficient for Business-to-Business (B2B) trade. Paying with these methods is not only cumbersome, but also inconvenient when it comes to keeping financial records up-to-date. For instance, in case of Debit/Credit cards, businesses have to acquire a separate Debit/Credit card for every employee who has the privilege to make a purchase on behalf of the company. Moreover, the buyer has to obtain a separate receipt with the detailed tax information of the purchased items from the seller. This tax information needs to be added manually in the financial/book-keeping system. On the other hand, paying by invoice is an optimal choice for businesses, because it solves the problems posed by the conventional payment methods.

Enterpay Oy provides an invoice-based B2B payment method, which enables buyers to make purchases on credit and takes the credit risk away from the sellers. From the perspective of banks and financial companies, Enterpay provides an online credit decision engine that helps them expand their business and offer a reliable payment service in the B2B market.

In this thesis, we identified that Enterpay's payment service is currently facing two major challenges, i.e., efficient scaling and the need to be flexible in the choice of technology stack for different components of the application. We observed that the current monolithic architecture of the application is rigid and is an obstacle for efficient scalability. Consequently, we proposed the migration to a microservice-based architecture as a solution.

We identified some of the modules from the original monolithic application that have different scalability criteria. We extracted out these modules and built them as independent microservices. We introduced Docker in the Enterpay's technology stack, so that each microservice can be built as a separate Docker image and can have a independent development and deployment life-cycle. Additionally, we selected Kubernetes as the container

orchestration environment for the microservice-based architecture. We created separate Kubernetes deployment and service resources, allowing each microservice to have its own replication strategy and load balancing capabilities. Next, we created an ingress resource which acts as a single entry point for the payment service. This component receives all user traffic and distributes the user requests to the relevant microservice.

Finally, we created a Kubernetes cluster on Amazon Web Service (AWS) and deployed the microservices. We formulated three different hypotheses to evaluate the performance of the microservice-based architecture. Accordingly, we conducted experiments to analyze the performance of microservice-based architecture in terms of scaling and recovering from instance failures. Moreover, we realized that microservice-based architecture introduces inter-service communication cost. Thus, we conducted an experiment to examine the impact of this added cost. For this experiment, We selected an application scenario, where 2 microservices need to communicate and collaborate with each other on a given task. We collected the measurements with different variations and calculated the standard deviation. After analyzing the measurements, we concluded the following

- The mircoservice-based architecture scales efficiently and recovers from instance failures faster than the old monolithic architecture.
- The average response time of service requests is comparable with the original monolithic architecture. The inter-service communication did not add a significant delay in the average response time of service requests.
- The microservice-based architecture allowed Enterpay to be flexible in the choice of technology for different new features it wants to build. We noticed that not only Enterpay was able to build one of the new features with different technology stack, but an existing API which originally needed a separate EC2 instance for its deployment, was successfully deployed in the same Kubernetes cluster.

8.1 Future work

At the end of this thesis, the new microservice-based architecture is deployed in the test environment. In near future, Enterpay will replace the old architecture by the new microservice-based architecture in production environment. With this adaptation of microservice-based architecture, Enterpay will adjust its internal practices of development, testing and deployment

accordingly. For instance, Enterpay will continue to benefit from software containers and each microservice will be built as a Docker container. Furthermore, the development team of Enterpay will extract more modules based on their scalability criteria and deploy them as independent microservices. New features will be developed with the best suited programming language and frameworks. These features will be deployed as microservices in the same kubernetes cluster on AWS.

Bibliography

[1] Iso/iec/ieee international standard - systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)* (Dec 2010), 1–418.

[2] Ahmad, B. Coordinating vertical and horizontal scaling for achieving differentiated qos. Master's thesis, University of Oslo, 2018.

[3] amazon.com. Auto Scaling Groups). `https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html`. Accessed 14 Dec 2019.

[4] amzon.com. Application Architecture. `https://docs.aws.amazon.com/AmazonECS/latest/developerguide/application_architecture.html`. Accessed 29 Oct 2019.

[5] amzon.com. What is Amazon Elastic Container Service? `https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html`. Accessed 29 Oct 2019.

[6] aquasec.com. Docker Architecture. `https://www.aquasec.com/wiki/display/containers/Docker+Architecture`. Accessed 12 Sep 2019.

[7] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev. 37*, 5 (Oct. 2003), 164–177.

[8] Boettiger, C. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev. 49*, 1 (Jan. 2015), 71–79.

[9] Chris Richardson. Pattern: API Gateway / Backends for Frontends. `https://microservices.io/patterns/apigateway.html`. Accessed 09 Nov 2019.

[10] docker-curriculum.com. Learn to build and deploy your distributed applications easily to the cloud with Docker. https://docker-curriculum.com/#introduction. Accessed 29 Mar 2019.

[11] docker.com. Docker Documentation. https://docs.docker.com/. Accessed 12 Sep 2019.

[12] docker.com. Docker overview. https://docs.docker.com/engine/docker-overview/. Accessed 12 Sep 2019.

[13] docker.com. Install Docker Desktop on Mac. https://docs.docker.com/docker-for-mac/install/. Accessed 09 Nov 2019.

[14] docker.com. Overview of Docker Compose. https://docs.docker.com/compose/. Accessed 09 Nov 2019.

[15] docker.com. Use multi-stage builds. https://docs.docker.com/develop/develop-images/multistage-build/. Accessed 12 Sep 2019.

[16] Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. DevOps. *IEEE Software 33*, 3 (May 2016), 94–100.

[17] Eder, M. Hypervisor-vs . container-based virtualization.

[18] Google.com. CONTAINERS AT GOOGLE. https://cloud.google.com/containers/. Accessed 29 Mar 2019.

[19] Graziano, C. D. *A performance analysis of xen and kvm hypervisors for hosting the xen worlds project.* PhD thesis, Iowa state university, 2011. https://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=3243&context=etd.

[20] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Slacker: Fast distribution with lazy docker containers. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Berkeley, CA, USA, 2016), FAST'16, USENIX Association, pp. 181–195.

[21] IBM developer. Learn about hypervisors, system virtualization, and how it works in a cloud environment, 2011. https://developer.ibm.com/articles/cl-hypervisorcompare/. Accessed 29 Mar 2019.

[22] Isaac Eldridge. What Is Container Orchestration? https://blog.newrelic.com/engineering/container-orchestration-explained/. Accessed 12 Sep 2019.

[23] Jaramillo, D., Nguyen, D. V., and Smart, R. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016* (March 2016), pp. 1–5.

[24] Jeff Barr. Amazon EC2 Container Service (ECS) â Container Management for the AWS Cloud. `https://aws.amazon.com/blogs/aws/cloud-container-management/`. Accessed 29 Oct 2019.

[25] Justin Ellingwood. An Introduction to Kubernetes. `https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes`. Accessed 12 Sep 2019.

[26] kubernetes.io. Getting Started - Container runtimes. `https://kubernetes.io/docs/setup/production-environment/container-runtimes/`. Accessed 29 Oct 2019.

[27] kubernetes.io. Install Minikube. `https://kubernetes.io/docs/tasks/tools/install-minikube/`. Accessed 09 Nov 2019.

[28] kubernetes.io. kube-proxy. `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/`. Accessed 29 Oct 2019.

[29] kubernetes.io. Kubernetes Cloud Controller Manager. `https://kubernetes.io/docs/tasks/administer-cluster/running-cloud-controller/`. Accessed 29 Oct 2019.

[30] kubernetes.io. ReplicationController. `https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#common-usage-patterns`. Accessed 29 Oct 2019.

[31] kubernetes.io. Secrets. `https://kubernetes.io/docs/concepts/configuration/secret/`. Accessed 09 Nov 2019.

[32] kubernetes.io. What is Kubernetes. `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`. Accessed 12 Sep 2019.

[33] learnitguide.net. What is Kubernets - Learn Kubernetes from Basics. `https://www.learnitguide.net/2018/08/what-is-kubernetes-learn-kubernetes.html`. Accessed 12 Sep 2019.

[34] Lingayat, A., Badre, R. R., and Kumar Gupta, A. Performance evaluation for deploying docker containers on baremetal and virtual machine. In *2018 3rd International Conference on Communication and Electronics Systems (ICCES)* (Oct 2018), pp. 1019–1023.

[35] Margaret Rouse. application containerization (app containerization). https://searchitoperations.techtarget.com/definition/application-containerization-app-containerization. Accessed 29 Mar 2019.

[36] Margaret Rouse. B2B (business-to-business). https://searchcio.techtarget.com/definition/B2B. Accessed 09 Nov 2019.

[37] Margaret Rouse. Type 2 hypervisor (hosted hypervisor). https://searchservervirtualization.techtarget.com/definition/hosted-hypervisor-Type-2-hypervisor. Accessed 29 Mar 2019.

[38] Margaret Rouse. hot swap, 2005. https://whatis.techtarget.com/definition/hot-swap. Accessed 26 Oct 2019.

[39] Margaret Rouse. Monolithic Architecture, 2016. https://whatis.techtarget.com/definition/monolithic-architecture. Accessed 29 Mar 2019.

[40] Mark Richards. *Microservices Antipatterns and Pitfalls.* O'Reilly Media, Inc., 2016.

[41] Martin Fowler. What are Microservices? https://www.martinfowler.com/microservices/#what. Accessed 29 Mar 2019.

[42] Martin Fowler. GOTO 2014 â Microservices â Martin Fowler, 2014. https://www.youtube.com/watch?v=wgdBVIX9ifA. Accessed 29 Mar 2019.

[43] microservices.io. Monolithic Architecture. https://microservices.io/patterns/monolithic.html. Accessed 29 Mar 2019.

[44] Microsoft. Resize virtual machines. https://azure.microsoft.com/en-us/blog/resize-virtual-machines/. Accessed 26 Oct 2019.

[45] mongodb.com. Containers and Orchestration Explained. https://www.mongodb.com/containers-and-orchestration-explained. Accessed 29 Oct 2019.

[46] Morabito, R., KjÃllman, J., and Komu, M. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering* (March 2015), pp. 386–393.

[47] OPENSOURCE.COM. What is virtualization? https://opensource.com/resources/virtualization. Accessed 29 Mar 2019.

[48] PAHL, C. Containerization and the paas cloud. *IEEE Cloud Computing 2*, 3 (May 2015), 24–31.

[49] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (July 1974), 412–421.

[50] PRIYANKA PRAKASH. B2B Payments: The 8 Best Payment Solutions for Your Business). https://www.fundera.com/blog/b2b-payments. Accessed 09 Nov 2019.

[51] PRIYANKA PRAKASH. Deducting VAT on purchases). https://www.vero.fi/en/businesses-and-corporations/about-corporate-taxes/vat/deducting-vat-on-purchases/. Accessed 09 Nov 2019.

[52] REKHA, G. S. A study on virtualization and virtual machines. *International Journal of Engineering and Science Invention 7*, 51–55.

[53] RENSIN, D. K. *Kubernetes - Scheduling the Future at Cloud Scale.* 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[54] RICHARD LI. Canary Deployments, A/B Testing, and Microservices With Ambassador. https://dzone.com/articles/canary-deployments-ab-testing-and-microservices-wi. Accessed 29 Mar 2019.

[55] SAHA, P., BELTRE, A., UMINSKI, P., AND GOVINDARAJU, M. Evaluation of docker containers for scientific workloads in the cloud. In *Proceedings of the Practice and Experience on Advanced Research Computing* (New York, NY, USA, 2018), PEARC '18, ACM, pp. 11:1–11:8.

[56] SAM NEWMAN. *Building microservices.* O'Reilly Media, Inc., 2015.

[57] SPRUHA PANDYA. Of Microservices Containers. https://hackernoon.com/https-medium-com-spruha-pandya-of-microservices-containers-6f0ea25dac3. Accessed 26 Oct 2019.

[58] TIFFANY JERNIGAN. Building Blocks of Amazon ECS. https://medium.com/containers-on-aws/building-blocks-of-amazon-ecs-db7fdfeeaa6f. Accessed 12 Sep 2019.

[59] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 18:1–18:17.

[60] Yourdon, E., and Constantine, L. L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 1st ed. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.

Appendix A

Implementation - Intermediate steps

A.1 docker-compose

After introducing Docker, we used *docker compose*, which is a tool suitable for running multi-container applications [14]. Docker compose comes with the Docker desktop package and can be used to run/stop multiple containers with a single command. It allows us to defines the configurations of application services in a YAML or JSON file. This file specifies all services which should be built and run within the application. Additionally, it contains instructions on how the containers should be built and run. Docker compose builds an image only if it is not already built. Consequently, We created a file named *docker-compose.yaml* as shown in File 3.

Once the docker-compose file is ready, we ran the following command, which sequentially builds the specified Docker images and finally runs the containers simultaneously.

```
 # reads the docker-compose.yaml and starts
 # all the containers together.
$ docker-compose up
```

In case, the user wants to stop all of the containers, he/she needs to run the following command and all of containers will be terminated together.

```
 # shuts down all containers
$ docker-compose down
```

```
version: 3.0
services:
  httpd:
    build:
      context: .
      args:
        - NODE_ENV=local
      dockerfile: Dockerfile_httpd
    ports:
    - 80:80/tcp
  main:
    build:
      context: .
      args:
        - NODE_ENV=local
      dockerfile: Dockerfile_main
    ports:
    - <main_port>:<main_port>/tcp
  connectin-main:
    build:
      context: .
      args:
        - NODE_ENV=local
      dockerfile: Dockerfile_connectin
    ports:
    - <connectin_port>:<connectin_port>/tcp
  ui:
    build:
      context: .
      args:
        - NODE_ENV=local
      dockerfile: Dockerfile_ui
    image: enterpayoy/thesis_ui
    ports:
    - <ui_port>:<ui_port>/tcp
  actors:
    build:
      context: .
      args:
        - NODE_ENV=local
      dockerfile: Dockerfile_actors
    ports:
    - <actor_port>:<actor_port>/tcp
```

File 3: docker-compose.yaml specifying different microservices.

A.2 Deployment on local Kubernetes cluster

First we deployed the microservices to a local Kubernetes cluster. Docker version 2.1.0.3 comes with built-in support for Kubernetes. We installed a tool named *minikube*, which is a handy tool to get started with Kubernetes on a developer's machine. Minikube installs the Kubernetes locally and creates a single node cluster. This cluster runs inside a virtual machine on developer's

computer. The installation instructions are available on Kubernetes official website [27].

We used the following commands to create the local Kubernetes cluster and deploy the deployment and service resources.

```
 # creates the single node local Kubernetes cluster
$ minikube start
 # deploys the configurations defined in main-service.yaml file
$ kubectl create -f main-service.yaml
 # enlists the running pods
$ kubectl get pod
 # prints the url for the specified service on console
$ minikube service main-service --url
 # deletes specified service
$ kubectl delete services main-service
 # stops the local Kubernetes cluster
$ minikube stop
 # deletes the local Kubernetes cluster
$ minikube delete
```

Additionally, we used the *Web UI* for Kubernetes, which is convenient interface for deploying/managing the Kubernetes services and resources. For example, this tool can be used to create, modify or scale the deployments[1].

```
 # runs the Web UI for the local cluster
$ minikube dashboard
```

A.3 Deployment on AWS Kubernetes cluster

We used a tool called *Kubernetes Operations (kops)* to create the kubernetes cluster on AWS. The detailed instructions on how to setup kubernetes cluster on AWS are available on the official git repository of kops[2]. As a prerequisite, we installed AWS Command Line Interface (CLI) tools.

By default, kops creates a three node cluster, where one server is master with the instance type of m3.medium and the rest of the two nodes are worker nodes with the instance type of t2.medium. The details of these server can be found on AWS's website [3].

Following is the list of commands that we used to to create the kubernetes cluster on AWS:

```
 # installs AWS CLI
$ brew update && brew install kops
 # create a bucket on Amazon's Simple Storage Service (S3). This bucket will contain
 # the cluster state metadata
```

[1] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

[2] https://github.com/kubernetes/kops/blob/master/docs/getting_started/aws.md

[3] https://aws.amazon.com/ec2/instance-types/

```
$ aws s3api create-bucket --bucket <bucket_name> --region <desired_region>
 # creates the required configuration for the cluster
$ kops create cluster --zones eu-central-1a <cluster_name>
 # builds the cluster, installs kubernetes components on all nodes
$ kops update cluster <cluster_name> --yes
 # deletes the cluster and all its components
kops delete cluster --name <cluster_name>
```

A.4 Kubernetes cluster resources

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: main
  name: main-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: main
  strategy: {}
  template:
    metadata:
      labels:
        app: main
    spec:
      containers:
      - image: <remote_image_name>
        name: main
        ports:
        - containerPort: <contianer_port>
        resources: {}
      restartPolicy: Always
      imagePullSecrets:
      - name: regcred
status: {}
----------------
apiVersion: v1
kind: Service
metadata:
  name: main-service
spec:
  type: NodePort
  ports:
  - name: <target_port_name>
    port: <target_port>
    targetPort: <target_port>
    protocol: TCP
    nodePort: <node_port>
  selector:
    app: main
```

File 4: main-service.yaml: Deployment and Service resource for Main Service