Yuhao Zhou

MICROSERVICES AND DEVOPS INTEGRATION

Technology and Communication
2024

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Dr. Ghodrat Moghadampour, for his invaluable guidance, support, and encouragement throughout the research process. His expertise, constructive feedback, and unwavering commitment have been instrumental in shaping this thesis and my academic journey.

I extend my appreciation to Vaasa University of Applied Science for providing the necessary resources and facilities for conducting this thesis.

Special thanks to my family for their unconditional love, understanding, and encouragement. Their unwavering support and belief in my abilities have been a constant source of motivation.

I would also like to acknowledge the contributions of my tech leader Chang Lijun, CEO of the company Shanghai Yushu Technology, for his collaboration and assistance during various stages of this research project.

Lastly, I am thankful to all my friends and colleagues for their encouragement, discussions, and moral support throughout this journey.

This thesis would not have been possible without the collective efforts and support of all those mentioned above. Thank you.

22 May 2024

Yuhao Zhou

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Cloud-based Software Engineering

ABSTRACT

Author	Yuhao Zhou
Title	Microservices and DevOps Integration
Year	2024
Language	English
Pages	62 + 6 Appendices
Name of Supervisor	Ghodrat Moghadampour

This thesis aimed to make an existing web application more flexible for developers to maintain, easier for operation staff to scale the resource when user demands increased and reduce the server errors experienced by users of the application.

A big, old application was broken into smaller pieces called microservices. These microservices were then deployed into a container orchestration environment for easier management. DevOps tools were also added to automatically test and deploy changes.

As a result, two new microservices were created using the Go programming language and were running in a MicroK8s environment. Istio was added to help manage communication between services. All microservices are integrated with GitHub Actions for automatic testing and deploying.

Now the development team can release updates more frequently, and the backend of the application can scale up without much manual work. Users are getting bug fixes and new features more often and are experiencing fewer backend errors than before.

In the future, the rest of the application can be converted into microservices by using the same methods in this thesis.

Keywords microservices, DevOps, service mesh, microK8s

CONTENTS

ABSTRACT

1 INTRODUCTION 1
1.1 Background 1
1.2 Motivation and Objectives 2
2 RELEVANT TECHNOLOGIES 4
2.1 Microservices Architecture 4
2.1.1 Background of Microservices 4
2.1.2 Benefits of Microservices 6
2.1.3 Challenges of Microservices 8
2.1.4 Building Microservices 9
2.2 DevOps 10
2.2.1 Background 11
2.2.2 Benefits of DevOps 11
2.2.3 DevOps Practices 12
2.3 Istio Service Mesh 14
2.4 GitHub Actions 16
2.5 Kubernetes 17
2.6 MicroK8s 19
2.7 The Go Programming Language 20
2.8 Docker 22
2.9 PostgreSQL 24
3 APPLICATION DESCRIPTION 26
4 DATABASE DESIGN 32
5 IMPLEMENTATION 36
5.1 Code Structure 37
5.2 MicroK8s 39
5.3 Istio Service Mesh 40

5.4 Load Balancing 41
5.5 GitHub Actions 42
6 TESTING 44
7 SUMMARY 48
8 CONCLUSIONS 50
REFERENCES 52
APPENDICES 55

LIST OF FIGURES AND TABLES

Figure 1. Use cases diagram of the mind mapping application of this thesis. 28

Figure 2. Sequence diagram of the mind mapping web application. 29

Figure 3. Sequence diagram of the CI/CD workflows of the application. 31

Figure 4. Class diagram of the application. 32

Figure 5. The result of GitHub Actions workflows. (Screenshot of Chrome browser) 44

Figure 6. Traffic tracking result of the service mesh. (Screenshot of Chrome browser) 46

Figure 7. Memory consumption of the host Linux server. (Screenshot of SSH in terminal) 47

Table 1. The objectives weightage table of this thesis. 26

Table 2. Priority table of requirements of this thesis. 27

Table 3. Achievement table of requirements in this thesis. 48

LIST OF CODE SNIPPETS

Code Snippet 1. Schema creation SQL of the template table in Template-API service. 34

Code Snippet 2. Schema creation SQL of the material table in Material-API service. 35

Code Snippet 3. Code structure of Template-API service. 37

Code Snippet 4. Code structure of Material-API service. 38

Code Snippet 5. Shell script of installing MicroK8s on Ubuntu. 39

Code Snippet 6. Shell script of enabling Istio service mesh in MicroK8s. 40

Code Snippet 7. Load balancing configuration of the microservices application. 41

Code Snippet 8. Configuration of building Docker image for the application. 42

Code Snippet 9. The terminal output of unit test result of the application......... 45

APPENDICES

APPENDIX 1. Code snippet of application configuration of Template-API service.

APPENDIX 2. Code snippet of application configuration of Material-API service.

APPENDIX 3. Code snippet of interservice communication of the application.

APPENDIX 4. Code snippet of workflows configuration of GitHub Actions.

1 INTRODUCTION

The company's product in this thesis is a mind mapping web application designed to help users organize their thoughts and manage ideas. It can be used to draw beautiful diagrams and flowcharts. Users can create and edit diagrams in web browsers. All the user data and files they created are stored in the cloud.

The product consists of two parts: frontend and backend. The frontend of the application is written in JavaScript. The backend of the application is written in Go programming language.

As time went by, the growing complexity of the application became a big problem for the development teams. Developers were struggling with the chaos of dependency entanglement and felt difficult to update the backend services. The process of releasing a new feature became slower and slower.

As the number of users grew, scalability issues kept happening, making the system unreliable from the users' perspective. During weekends, there was a spike access to the backend and caused ephemeral error related to resources shortage.

The development teams are longing for a solution to these problems. There was a strong need to improve the backend of the application.

1.1 Background

The Mind Map web application of the company started out as a simple tool to help users organize their thoughts and manage them efficiently. With an intuitive interface, it allows users to create and edit beautiful diagrams and flowcharts directly in the web browser, with all data stored securely in the cloud. This seamless experience quickly attracted a growing user base.

As the application grew, users noticed issues that impacted their experience. System reliability became a concern, with frequent outages and performance issues, especially during peak hours. These issues were most noticeable during weekend

peak periods, when resource shortages caused brief errors and service unavailable. Additionally, users were seeing a significant slowdown in the delivery of new features. The once responsive development process seems to be bogged down, resulting in longer wait times for updates and improvements.

From the developers' point of view, the increasing complexity of the application was a major hurdle. The backend, originally designed to handle a modest load, struggled to scale with the growing number of users. The architecture, initially manageable, had evolved into a monolithic structure where components were tightly coupled. This dependency entanglement made it difficult to implement changes without affecting other parts of the system.

The scalability issues were exacerbated by the old monolithic architecture, which did not lend itself well to the dynamic and expanding demands of the user base. Furthermore, many processes within the development and deployment pipeline remained manual. These manual tasks not only consumed significant time and effort but also introduced the potential for human error, further slowing down the release of new features and patches.

1.2 Motivation and Objectives

To address the challenges posed by the monolithic architecture, migrating to a microservices architecture is imperative. This approach allows the application to be broken down into smaller, independent services that can be developed, deployed, and scaled individually. This modularity enhances system reliability by isolating failures and enables faster delivery of new features by reducing dependency entanglement. Alongside this architectural shift, adopting DevOps practices is crucial. DevOps fosters a culture of collaboration between development and operations, streamlines workflows through automation, and ensures continuous integration and delivery. This combination of microservices and DevOps will not only resolve current scalability and complexity issues but also position the development team to respond more agilely to user needs and market changes.

The thesis has 3 objectives:

1. Faster feature delivery. By decoupling services, development teams can work on multiple parts of the application simultaneously without being hindered by interdependencies. By integrating CI/CD pipelines, many manual workflows became automated. Those two approaches lead to faster feature delivery.
2. Scalability. Enable the application to scale more effectively in response to increasing user demands and workload fluctuations. By using microservices architecture, selective scaling of the application is possible.
3. Reliability. Increase the resilience of the system by isolating faults within individual microservices.

As a result, a decision was made to use Microservices architecture to fix the problems above and use DevOps practices to improve the agility of the development team.

The topic of this thesis explores the integration of microservices architecture with DevOps practices to enhance software development and deployment processes. It focuses on how breaking down a monolithic application into smaller, independent services can improve scalability, reliability, and feature delivery. Additionally, the thesis examines how adopting DevOps principles, such as continuous integration, continuous delivery, and automation, can streamline workflows, foster collaboration between development and operations teams, and lead to a more agile and resilient system.

2 RELEVANT TECHNOLOGIES

This section provides an overview of the key tools, frameworks, and methodologies that play a significant role in the integration of microservices and DevOps practices. This section serves to familiarize the reader with the foundational technologies underpinning the proposed approach, laying the groundwork for subsequent discussions on implementation and analysis. By examining these technologies in context, readers will gain a comprehensive understanding of the ecosystem necessary for effective adoption and integration within software development projects.

2.1 Microservices Architecture

Microservices architecture is “popular architectural style for building applications that are resilient, highly scalable, independently deployable, and able to evolve quickly” (Microsoft Azure, 2024, p. 1). It is becoming more and more popular in software engineering area for its ease to scale and speed to develop, especially in this rapidly evolving world, being agile is the key to success nowadays.

2.1.1 Background of Microservices

Before microservices architecture emerged, software was mostly written in monolithic architecture style. With monolithic architectures, all processes are tightly coupled and run as a single service. This means that if one process of the application experiences a spike in demand, the entire architecture must be scaled. (Amazon Web Services, 2024, p. 1) This brings unnecessary resource waste as the other part of the application does not need that much of resource to run. As the code base grows, the dependencies inside the application entangled gradually tighter unavoidably, which makes adding and changing functionalities to the overall application increasingly complex. This complexity would eventually limit the ability of experimentation and implementation of innovative ideas in the R&D department which slows the pace the product development of the company, even worse

when the company is in a crucial phase of developing innovative ideas to save the company from bankruptcy. Monolithic architecture increases application availability risks because too many dependent and tightly coupled processes will increase the impact of a single process failure. (Ponce, et al., 2019, p. 1)

But in a microservices architecture, applications are divided into multiple independent small services which are designed business unrelated as much as possible. Those services communicate through well-defined interfaces using lightweight APIs. (Ponce, et al.,2019, pp. 1-2)

The evolution of computing technologies has been significantly influenced by the varying requirements of business applications. Traditionally, applications were developed in a monolithic style, residing on a single machine, and typically operated by a single user. However, the advent of networking technologies introduced the concept of sharing computing resources (both hardware and software) among different users over a network. This led to the development of splitting applications into identifiable tiers: clients and servers. (Surianarayanan, et al., 2019, pp. 2-3)

In this model, the client tier usually consists of a physical machine running the client or interface part of the application, while the service tier runs the core application on a separate physical machine. Users can access applications on servers over the network, a setup known as "two-tier client-server programming." This marked a shift from monolithic to network-based computing, enabling simultaneous access by many users. Initially, two-tier client-server applications were accessed over a Local Area Network (LAN), supporting about ten users. However, as user numbers grew, the LAN-based two-tier model became inadequate for scalability and availability. (Surianarayanan, et al., 2019, pp. 2-3)

To address these limitations, the two-tier model was enhanced into a three-tier architecture, separating business logic from databases. The three logical modules client/presentation, business logic, and data access/database—were deployed on

different machines. This three-tier architecture became the standard business model for over two decades. (Surianarayanan, et al., 2019, pp. 2-3)

The evolution of the internet brought significant changes to business applications. Users wanted to access applications over the internet, leading to the development of web applications. This shift not only changed access methods but also significantly increased the number of users for commercial applications such as online shopping, travel booking, banking, financial services, and e-governance. (Surianarayanan, et al., 2019, pp. 2-3)

These trends demanded that business applications be readily available and highly scalable, prompting further changes in application development. The expansion of networks meant that applications were distributed over long-range networks. Enterprises worldwide began developing distributed applications, with components distributed across different locations and accessed over private networks. Concurrently, object-oriented distributed programming technologies were developed to handle the distribution and interaction of objects between remote machines. (Surianarayanan, et al., 2019, pp. 2-3)

To meet the dynamic needs of customers, businesses needed to integrate individual applications for efficient information aggregation. This led to the adoption of Service-Oriented Architecture (SOA), typically implemented using XML-based web services, to facilitate open communication protocols. (Surianarayanan, et al., 2019, pp. 2-3)

2.1.2 Benefits of Microservices

The benefits of migrating from traditional monolithic architecture into microservices architecture are many. It solved most of the problems that traditional monolithic architecture faces in modern software development industry. It meets the key requirements of the current software development. Here are the descriptions of what microservices could bring to the development teams.

Microservices architecture helps improve agility of development for the team to experiment and adopt new ideas. When using monolithic architecture, the process of releasing a new feature or bug fix can be tedious and time wasting as the code base grows and one bug fix update can block the entire release process for a long time. While microservices architecture solved this problem by dividing the application into multiple independently deployable small services. (Ponce, et al., 2019, p. 2)

Small and focused teams help avoid communication chaos. When teams are huge, it is very easy for each team member to lose the understanding of their role and responsibilities. These misunderstandings might eventually lead to an inefficient communication flow in the team will probably make the whole company evolve slower and slower. Small teams usually do not have those problems, each member in the small team can easily have a clear mindset of what their job is. (Ponce, et al., 2019, p. 2)

The small code base helps reduce the dependencies entanglement. When using the traditional monolithic architecture, it is quite easy for the team members to make the code dependencies tangled as the code base grows. The more team members and code dependencies involved when adding or updating a code change to the code base, the longer the releasing process would take. Microservices architecture solved this issue. (Amazon Web Services, 2024, p. 1)

The decomposition of microservices architecture brings fault isolation to the system. When using a monolithic architecture, it is very common to have one part of the application fail that causes the entire application unavoidable. However, in the microservices architecture, services are designed independent individually, single service's failure will not cause the entire system crash. (Ponce, et al.,2019, p. 2)

The decomposition of microservices architecture also brings scalability to the system. Monolithic architecture cannot scale specific parts of the application. But in

microservice architecture, applications are divided into small, independent and domain specific microservices which makes possible to scale up the single microservice which is in high demand to customers and the rest of the microservices remains the same. (Balalaie, et al.,2016, p. 2)

2.1.3 Challenges of Microservices

There are also downsides of microservices architecture that developers might need to think about when considering migrating a traditional monolithic architecture into microservices architecture. This paragraph describes some of the common challenges that development teams would face when using a microservice architecture.

In a microservices architecture, each service can be simpler compared to the traditional monolithic application, but the entire system, as the amount of microservices grows, becomes more and more complex. The development teams must seek a better tool to help them manage all these microservices, and that tool is usually called service mesh. (Istio Team, 2024, p. 1)

Developing and testing an application that has complex cross-microservices communication can be difficult. It is not as easy as developing and testing the traditional monolithic application where all the logic is running inside the same process which has a mature ecosystem to debug and tracing the failure. Applications that are in microservices architecture must have something like service mesh and distributed tracing framework to debug and test the application which is much more complex than the monolithic way. It is also very challenging to handle the dependency issues when the application is evolving quickly. (Istio Team, 2024, p. 1)

The distributed way of building microservices applications has advantages in flexibility and agility, but it can also lead to skill set problems. The moment when different teams use different programming languages and technologies to build their own microservices, it becomes harder and harder for the tech leaders to control

the technology stack to keep the quality of the software stay consistent. Besides the problem of mix of technologies, the necessary technologies along with microservices architecture, such as service mesh and container orchestrators, also requires a lot of skill set, knowledge and talents in the company to handle potential issues which can be a considerable re-source overhead. (Balalaie, et al., P,2016, p. 14)

Compared to traditional monolithic architecture where threads communicate through memory, network congestion and latency could be another challenging problem to handle when using microservices architecture as the inter-service communication approach in microservices architecture is using network connection. Especially when the callback chain of service dependencies gets too long, the extra network latency could become a considerable loss. (Microsoft Azure, 2024, p. 1)

2.1.4 Building Microservices

To design a good microservices architecture, proper tools and analysis approaches are needed. A well-defined microservice architecture can prevent most of the common pitfalls from happening.

Identifying the boundaries of each microservices is the first step of designing microservices. This step finalizes the de-sign of individual microservices. The graininess of microservices design determines the efficiency of the system and the development team. It is a bad practice to make the service too small or too big. It may add extra architecture complexities, resource overhead and operational complexity to the system if the system gets too fine-grained. Picking a good balance between them is the key to designing an efficient microservices architecture. (Newman, S, 2015, p. 2)

After the microservices system is deployed to the production environment, it is time to have developers or operators to operate, configure and monitor it. This

step requires the operator to have a strong skill set to set up everything and make sure the system is running correctly both in the present and in the future. (Newman, 2015, pp. 103-104)

2.2 DevOps

DevOps is the “combination of cultural philosophies, practices, and tools that increases an organization’s ability to deliver applications and services at high velocity” (Amazon Web Services, 2024, p. 1). Better efficiency and faster delivery of product value help organizations win in this rapidly evolving market.

DevOps integrates the worlds of development and operations through automated development, deployment, and infrastructure monitoring. It represents an organizational shift from siloed groups performing functions separately to cross-functional teams working on continuous operational feature deliveries. This approach helps deliver value faster and continuously, reduces problems due to miscommunication between team members, and accelerates problem resolution. (Ebert, et al., 2016, p. 1)

DevOps involves a cultural shift toward collaboration between development, quality assurance, and operations. Organizations set up continuous delivery with small upgrades. Companies such as Amazon and Google have led this approach, achieving cycle times of minutes. However, the achievable cycle time depends on environmental constraints and the deployment model. A single cloud service is easier to facilitate than actual software deliveries of real products. (Ebert, et al., 2016, p. 1)

DevOps can be applied to various delivery models but must be tailored to the environment and product architecture. Not all products facilitate continuous delivery, such as in safety-critical systems. Nevertheless, even in constrained environments, upgrades can be planned and delivered quickly and reliably, as demon-

strated by the recent evolution of automotive software over-the-air updates. Besides highly secured cloud-based delivery, such models need dedicated architecture and hardware changes. One example is a hot-swap controller where one half is operational while the other half builds the next updates, which are swapped to active mode after in-depth security and verification procedures. DevOps for embedded systems is more challenging than for cloud and IT services because it attempts to combine legacy code and architecture with continuous delivery. (Ebert, et al., 2016, p. 1)

2.2.1 Background

What are the effects of a software delivery process on the participants, and why does it lead to conflict? As more features are completed, the developer's reputation improves. Throughput and good velocity are considered reflections of great performance by the developers. In many situations, from the developer's viewpoint, the new features available on test machines are indistinguishable from the features deployed on production systems available for users. (Httermann, 2012, p. 21)

Programmers, testers, database administrators, and system administrators experience challenges every day. These problems include risky or faulty deployments of software, an unnecessarily sluggish delivery process, and suboptimal collaboration and communication due to silos. These issues often lead to an overall slowdown that causes the company to lag its competitors and thus be placed at a disadvantage. (Httermann, 2012, p. 21)

2.2.2 Benefits of DevOps

There are a lot of benefits to applying DevOps model into the development teams.

DevOps model can significantly improve the response time to the market. By streamlining all the processes during software development, DevOps enables

faster releasing speed of features, updates, bug fixes and new products. The processes of testing, deployment and monitoring are automated by the practices of Continues Integration and Continues Delivery (CI/CD), which tremendously help the developers focus on the code itself instead of wasting energy on repetitive manual operations. (Amazon Web Services, 2024, p. 1)

The DevOps model enhances the quality of the software. By using continues testing and integration, most of the bug and obvious issues of the code can be caught earlier than the traditional way. (Amazon Web Services, 2024, p. 1)

DevOps can increase the efficiency of the development teams significantly. The key idea of DevOps principle is to automate all the repetitive works as much as possible. This helps the team focus more on delivery value to the customers, and makes the organization work more efficient and productive. (Amazon Web Services, 2024, p. 1)

DevOps can improve collaboration by promoting ownership and accountability. The development teams and operations teams work more closely than before. This reduces the inefficient communication between developers and operators in the traditional way. (Amazon Web Services, 2024, p. 1)

2.2.3 DevOps Practices

Treating operations (Ops) as first-class citizens from the point of view of requirements is crucial. Operations have specific needs related to logging and monitoring, and involving Ops in the development of requirements ensures that these needs are considered. For instance, logging messages should be understandable and usable by an operator. By including operations in the requirements development process, these requirements are more likely to be integrated effectively, contributing to overall high quality. (Bass, et al., 2015, pp. 4-5)

Making development (Dev) more responsible for incident handling can significantly shorten the time between the observation of an error and its repair. Organizations that implement these practices often have a period during which Dev is primarily responsible for a new deployment. After this initial period, responsibility shifts to Ops. This approach helps ensure that errors are addressed promptly, leveraging the expertise of the development team during the early stages of deployment. (Bass, et al., 2015, pp. 4-5)

Enforcing a standardized deployment process for both Dev and Ops personnel is essential for maintaining high-quality deployments. This practice helps avoid errors caused by ad hoc deployments and misconfigurations. A consistent deployment process also makes it easier to trace the history of a particular deployment artifact and understand the components included. This traceability is crucial for diagnosing and repairing errors efficiently. (Bass, et al., 2015, pp. 4-5)

Continuous deployment practices aim to reduce the time between a developer committing code to a repository and the code being deployed. This approach emphasizes the use of automated tests to increase the quality of code that makes its way into production. By streamlining the deployment pipeline, continuous deployment helps ensure that new features and fixes are delivered to users more rapidly and reliably. (Bass, et al., 2015, pp. 4-5)

Developing infrastructure code, such as deployment scripts, with the same practices used for application code is vital. This ensures both high quality in the deployed applications and that deployments proceed as planned. Errors in deployment scripts, such as misconfigurations, can lead to application, environment, or process errors. Applying quality control practices from software development to operations scripts and processes helps maintain the quality and reliability of these critical components. (Bass, et al., 2015, pp. 4-5)

A good implementation of DevOps model must include a well-designed monitoring and logging system. Advocating developers to focus more on their code itself does

not mean that developers do not have to care about the status after the code being pushed to the Git repository. Monitoring and logging refer to the importance of being responsible for the code that developers pushed to the repository and the experiences impact for users every time developers made a change to the code base. The system needs to be responsive, which means when unexpected issues happen, there should be a quick way to locate the source of the problems and being quick to solve it as soon as possible. (Amazon Web Services, 2024, p. 1)

2.3 Istio Service Mesh

Service meshes are rapidly becoming the standard component of cloud application in microservices architecture. A survey of the Cloud Native Computing Foundation (CNCF) community found that 68% of organizations are already using or planning to use service meshes in the next 12 months. The in-production use of service meshes has been growing 40-50% annually. Service meshes are popular because they solve important problems related to communication among loosely coupled microservices, the dominant paradigm for modern cloud applications. This includes discovering where services are located, establishing secure connections, and handling communication failures. They also offer many advanced capabilities such as rate limiting, load balancing, and telemetry via built-in or custom message processing filters. (Zhu, et al., 2022, p. 1)

However, service meshes are not without downsides. A primary one is overhead. All application traffic traverses software proxies, called sidecars, which increases request latency and consumes more resources. Service meshes can add tens of milliseconds to request latency in some settings and can consume multiple (virtual) CPU cores even at moderate load. These overheads can degrade user experience, increase operational costs, and decrease revenue. (Zhu, et al., 2022, p. 1)

By using a proxy on top of each microservice, service mesh becomes by far the industrial standard solution for solving the common problems that are brought by the architecture of microservices. Service mesh can provide observability, traffic

management, and security to the microservices architecture. The mechanism of adding a proxy layer in front of microservices moves the responsibility for these jobs away from the services themselves, so that developers do not need to change their code to have those powerful capability to the microservices architecture. (Istio Team, 2024, p. 1)

Istio is one of the popular solutions to service mesh. As the challenges that brought by migrating into microservices architecture described in the previous context, service mesh technology emerged to solve those problems.

The architecture of Istio service mesh consists of two main components: Istio control plane and Envoy proxy. Istio service mesh uses Envoy proxy as the communication gateway for each container inside the pods, which makes it possible to implement a service mesh layer transparently without making any code changes to the existing distributed application. Istio control planes are responsible for storing data and configurations centrally for the mesh. (Istio Team, 2024, p. 1)

This Envoy proxy manages traffic of the communication between microservices. The features for its traffic management include traffic routing, load balancing, service-to-service authentication, and traffic monitoring. It provides solutions to the common demands of microservices architecture, such as circuit breaking, timeouts, retries, service discovery, A/B testing and canary deployment. (Istio Team, 2024, p. 1)

As the mesh grows in complexity, it starts to be challenging to have a clear view on the call flow inside the mesh and the performance of the system. So, observability is also a key focus of Istio service mesh. It generates detailed telemetry for all the services in the mesh. Service metrics of four main signals are generated automatically for monitoring purposes: latency, traffic, errors, and saturation. It can be checked within the dashboard in the service mesh. It helps the operation teams get a basic insight on the performance of the mesh. (Istio Team, 2024, p. 1)

Distributed tracing log and access logs for each request can also be found in-side the dashboard of Istio service mesh. It provides detailed information of the communication flows inside the service mesh. (Istio Team, 2024, p. 1)

Istio is “a service mesh that was originally developed by Google but is now open source. It provides a way to connect, manage and secure microservices that communicate with each other” (Istio Team, 2024, p. 1). Istio is used in production by many companies such as Adobe, Baidu, and Google.

2.4 GitHub Actions

Automating repetitive tasks in the software development process is frequently supported by social code platforms, such as GitHub. GitHub Actions is a service offered by GitHub to automate all software workflows, including building, testing, and deploying directly from GitHub. GitHub Actions is relatively new, with a beta release available in November 2019. Since then, GitHub Actions has become a central service for both practitioners and cloud adoption. Software developers have a positive perception of GitHub Actions. (Valenzuela, Toledo, P., & Bergel, A., 2022, p. 1)

Despite the relevance of GitHub Actions in state-of-the-art software development practices, little is known about how practitioners build and maintain GitHub Actions workflows. It is not clear how practitioners cope with the particularities of developing GitHub Actions workflows. For example, workflows are executed only by pushing a change to the repository, debugging a GitHub Actions workflow is carried out by inspecting logs, and workflows are typically edited through a generic text editor in the GitHub interface. Developers can create a workflows manifest file inside the code repository which consists of building and testing, maybe deploying processes of configuration, and can be triggered conditionally when certain branch of the repository is merged or pushed. This allows developers to manage CI/CD configuration conveniently. It also supports features other than DevOps practice. It can help manage open-source issues opened by other developers by

automatically adding labels to the issues. It can increase the efficiency of development teams significantly. (GitHub, 2024, p. 1)

GitHub Actions provides multiple operating systems for virtual machines to run the workloads that developers defined, such as Linux, Windows and MacOS (Decan, et al., 2022, p. 8). When a pull request is merged or a certain branch is pushed, GitHub Actions will check whether the event meets the condition filter configured in the workload's definition file. Then GitHub Actions will run the workloads sequentially in the order defined by developers. GitHub Actions also provides workloads templates that developers can use to simplify the processes of writing workloads. (GitHub, 2024, p. 1)

Compared to other CI/CD tools like Jenkins, GitHub Actions is a new entrant in the competition of CI/CD market. It is becoming more and more popular among developers who use GitHub. (Decan, et al., 2022, p. 7)

2.5 Kubernetes

Years ago, most software applications were big monoliths, running either as a single process or as a small number of processes spread across a handful of servers. These legacy systems are still widespread today. They have slow-release cycles and are updated relatively infrequently. At the end of every release cycle, developers package up the whole system and hand it over to the operations (Ops) team, who then deploy and monitor it. In case of hardware failures, the ops team manually migrates it to the remaining healthy servers. (Luksa, 2017, pp. 1-3)

Today, these big monolithic legacy applications are slowly being broken down into smaller, independently running components called microservices. Because microservices are decoupled from each other, they can be developed, deployed, updated, and scaled individually. This enables quick and frequent changes to components to keep up with today's rapidly changing business requirements. (Luksa, 2017, pp. 1-3)

However, with a larger number of deployable components and increasingly larger data centres, it becomes more difficult to configure, manage, and keep the whole system running smoothly. It is much harder to figure out where to place each of those components to achieve high resource utilization and thereby keep hardware costs down. Doing all this manually is hard work. We need automation, which includes automatic scheduling of those components to our servers, automatic configuration, supervision, and failure handling. This is where Kubernetes comes in. Kubernetes enables developers to deploy their applications themselves and as often as they want, without requiring assistance from the operations team. But Kubernetes does not benefit only developers. It also helps the ops team by automatically monitoring and rescheduling those apps in the event of a hardware failure. The focus for system administrators shifts from supervising individual apps to mostly supervising and managing Kubernetes and the rest of the infrastructure, while Kubernetes itself takes care of the apps. (Luksa, 2017, pp. 1-3)

Kubernetes abstracts away the hardware infrastructure and exposes your whole data centre as a single enormous computational resource. It allows developers to deploy and run software components without having to know about the actual server's underneath. When deploying a multi-component application through Kubernetes, it selects a server for each component, deploys it, and enables it to easily find and communicate with all the other components of the application. (Luksa, 2017, pp. 1-3)

This makes Kubernetes great for most on-premises data centres, but where it starts to shine is in the largest data centres, such as those built and operated by cloud providers. Kubernetes allows them to offer developers a simple platform for deploying and running any type of application while not requiring the cloud provider's own sysadmins to know anything about the tens of thousands of apps running on their hardware. With more and more big companies adopting the Kuber-

netes model as the best way to run apps, it's becoming the standard way of running distributed applications both in the cloud and on local on-premises infrastructure. (Luksa, 2017, pp. 1-3)

2.6 MicroK8s

MicroK8s is a lightweight distribution of Kubernetes, developed by Canonical who own the Ubuntu operating system. It is an open-source system for automating deployment, scaling, and management of containerized applications. It provides the functionality of core Kubernetes components, in a small footprint, scalable from a single node to a high-availability production cluster. It is designed to be able to run most of the Kubernetes functionalities in low devices, such as Raspberry Pi, personal laptop, or virtual machines in the public cloud, with minimal resources consumption. It is promoted to be zero-ops, production-ready Kubernetes distribution. (MicroK8s, 2024, p. 1)

MicroK8s is very easy to install, compared to other lightweight distribution of Kubernetes. It can be installed on Ubuntu operating system by simply using one Snap installation command. This makes it incredibly easy to set up a new Kubernetes environment for development, testing and production usage. (MicroK8s, 2024, p. 1)

“Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications” (Kubernetes, 2024, p. 1). Compared to a full-fledged Kubernetes cluster, MicroK8s consumes minimal resources of computer as the name MicroK8s indicates. It makes MicroK8s very suitable for running on those resource-constrained devices, such as personal laptops and edge devices (Böhm, et al., 2021, p. 8).

MicroK8s was initially designed for developer workstations and was only later optimized for a low memory footprint. It prioritizes easy extensibility over minimal resource usage, making it a preferred choice when using Ubuntu and the Snap

package manager, although it can also be installed on Windows. High availability in MicroK8s is automatically activated on clusters with three or more nodes. Patch release updates are installed automatically in MicroK8s. (Koziolek, et al., 2023, p. 26)

2.7 The Go Programming Language

Go is an open-source programming language initiated by Google in 2009. Go is renowned for its robust support for system programming and its channel-based concurrency mechanism. It is advertised as "an open-source programming language that makes it easy to build simple, reliable, and efficient software." These strengths have made it the language of choice for many platforms software such as Docker and Kubernetes, which are the most common software for containerization management. With the growing popularity of containerization technology in today's software industry, Go has therefore become a key element of many modern software solutions. (Dilley, et al., 2019, p. 1)

The native inter-thread synchronization mechanisms in Go differ from more traditional synchronization mechanisms over shared memory by promoting the motto "don't communicate by sharing memory, share memory by communicating," encouraging communication via channels. Go is cloud native oriented. Go code can be compiled into a standalone binary executable that can be deployed without any runtime dependencies. The feature simplifies the process of deployment and can avoid many dependencies issues happening in the deployment phase. Go is designed to be simple, composable, and concurrent, which aligns well with the architecture of container orchestration tools. This makes Go a perfect choice for building applications in the cloud-native environment. (Andrawos, et al., 2017, p. 8)

The aims of the Go programming language are to be expressive, fast, efficient, reliable, and simple to write. Some programming languages, such as C or C++, are fast and reliable but not simple. Conversely, other programming languages, such

as Java or Python, are simple to write but not as efficient. Go is like the C programming language in many ways and is sometimes referred to as a “C-like language” or “C for the 21st century.” However, Go is much more than that, as it adopts good ideas from many other programming languages while avoiding features that lead to complexity or unreliability. (McGrath, 2020, p. 2)

Perhaps most importantly, Go introduces the ability to take advantage of multi-core CPU processing for concurrency using "goroutines" and "channels." This provides the potential for the computer to deal with several tasks at the same time. Although the Go language does not have the class structures found in Object-Oriented Programming (OOP) languages, such as C++ or Java, its features do provide some degree of encapsulation, inheritance, and polymorphism—the three cornerstones of OOP. (McGrath, 2020, p. 2)

2.8 Docker

Attention to cloud computing is increasing. Numerous technologies, such as Xen, Hyper-V, VMware vSphere, and KVM, have been developed by the IT industry and are known as virtualization technologies. To deploy many applications on the same virtual machine, applications and their dependencies need to be organized and isolated. Virtualization allows multiple applications to run on the same physical hardware. However, there are drawbacks to virtualization techniques: virtual machines are large, performance can be unstable due to running multiple virtual machines, the boot-up process is lengthy, and virtual machines struggle with issues like manageability, software updates, and continuous integration/delivery. (Potdar, et al., 2020, p. 1)

Docker is a containerization platform that streamlines the process of building, shipping, and running applications across various environments. By standardizing the deployment process, Docker has significantly simplified a previously complex development pipeline, which relied on diverse technologies such as virtual machines, configuration management tools, package management systems, and intricate library dependencies. (Miell, 2019, pp. 2-3)

Prior to Docker, these tools required specialized management and unique configurations, often resulting in fragmented workflows. Docker's introduction has unified these processes, enabling engineers to collaborate more efficiently by using a common pipeline and producing a single output deployable on any target system. This innovation has not only reduced the need for maintaining multiple tool configurations but has also established Docker as the standard solution for one of software development's most challenging aspects: deployment. Docker's rapid evolution and widespread adoption underscore its critical role in modern software engineering. (Miell, 2019, pp. 2-3)

Docker offers a compelling alternative to virtual machines (VMs) in many scenarios, particularly when the focus is on the application rather than the operating system.

By offloading OS management, Docker simplifies the developer's responsibilities. It outperforms VMs in several key areas: it spins up faster, is more lightweight for mobility, and facilitates easier and quicker sharing of changes through its layered filesystem. Docker's strong command-line integration and scripting capabilities further enhance its appeal. (Miell, 2019, pp. 7-8)

Docker excels in software prototyping by providing an isolated environment almost instantaneously. This allows developers to experiment without disrupting existing setups or enduring the complexities of provisioning a VM. The convenience and speed Docker offers in this regard can be truly transformative. (Miell, 2019, pp. 7-8)

Docker is also ideal for packaging software, especially for Linux users. A Docker image can run on any modern Linux machine without dependencies, akin to Java but without needing a JVM. This ensures consistent deployment across different environments, making Docker a versatile and powerful tool for both development and production. (Miell, 2019, pp. 7-8)

Docker helps to build and deploy containers, which can be used to package your applications and services. Containers are launched from images and can contain one or more running processes. Images can be thought of as the building or packaging aspect of Docker, while containers represent the running or execution aspect of Docker. (Combe, et al., 2016, p. 12)

Containerization is a technology that virtualizes applications in a lightweight manner, leading to significant adoption in cloud application management. A central challenge has emerged around orchestrating the construction and deployment of containers, both individually and in clusters. (Pahl, et al., 2017, p. 1)

Containers are an old concept. For decades, Unix systems have had the chroot command, which provides a simple form of filesystem isolation. Since 1998, FreeBSD has had the jail utility, which extended chroot sandboxing to processes.

Solaris Zones offered a comparatively complete containerization technology around 2001 but was limited to the Solaris OS. (Mouat, 2015, p. 5)

Then Google started the development of CGroups for the Linux kernel and began moving its infrastructure to containers. The Linux Containers (LXC) project started in 2008 and brought together CGroups, kernel namespaces, and chroot technology (among others) to provide a complete containerization solution. Finally, in 2013, Docker brought the final pieces to the containerization puzzle, and the technology began to enter the mainstream. (Mouat, 2015, p. 5)

Docker took the existing Linux container technology and wrapped and extended it in various ways. Primarily through portable images and a user-friendly interface—to create a complete solution for the creation and distribution of containers. The Docker platform has two distinct components: the Docker Engine (responsible for creating and running containers) and the Docker Hub (a cloud service for distributing containers). (Mouat, 2015, p. 5)

Containerization provides cloud application management based on lightweight virtualization. The increasing interest in cloud container technologies highlights the importance of their management and orchestration. (Pahl, et al., 2017, p. 13)

2.9 PostgreSQL

PostgreSQL is an advanced Object-Relational Database Management System (ORDBMS) with a rich history dating back to 1977. Originally developed as the Ingres project at the University of California, Berkeley, it evolved commercially through Relational Technologies, Ingres Corporation. In 1986, Michael Stonebraker's team at Berkeley extended the Ingres code to create Postgres, an object-relational database. By 1996, Postgres was renamed PostgreSQL, reflecting its enhanced functionality and a new open-source initiative. Today, PostgreSQL remains

actively developed by a global community of open-source contributors and is renowned for being the most advanced open-source database system. (Drake, et al., 2002, p. 1)

PostgreSQL's open-source nature allows users to freely obtain, use, and modify the software without proprietary restrictions. This transparency extends to performance benchmarks and statistics, which are often restricted by commercial database vendors like Oracle. The freedom to tailor PostgreSQL to specific needs is a significant advantage. However, a common misconception is that open-source software is always free of cost. While PostgreSQL can be downloaded without external costs, there may still be expenses related to deployment, maintenance, and support within a company. In summary, PostgreSQL's robust feature set and open-source foundation make it a powerful and flexible choice for database management, suitable for both small-scale applications and enterprise-level deployments. (Drake, et al., 2002, p. 2)

PostgreSQL is an open-source, client-server, relational database. It offers a unique mix of features that compare well to major commercial databases such as Sybase, Oracle, and DB2. One of the major advantages of PostgreSQL is that it is open source. People can see the source code for PostgreSQL. It is not owned by any single company; it is developed, maintained, broken, and fixed by a group of volunteer developers around the world. People don't have to buy PostgreSQL, it's free. People will not have to pay any maintenance fees (although commercial sources can be found for technical support). (Douglas, et al., 2003, p. 1)

PostgreSQL is highly extensible. Developers can build new functions, new operators, and new data types in the language of your choice. It is built around a client-server architecture, allowing building of client applications in several different languages, including C, C++, Java, Python, Perl, and others. On the server side, PostgreSQL includes a powerful procedural language. (Douglas, et al., 2003, p. 1)

3 APPLICATION DESCRIPTION

This section explains the details of the application built in this thesis. It includes a priority table of objectives and requirements, a use-case diagram of the application, a class diagram of the database design, and a sequence diagram of the CI/CD processes. With these figures and tables, readers can easily understand the application and follow along with the next implementation chapter.

Table 1 shows the three main objectives of this thesis and their weightages, smaller numbers in the table indicate lower weightages. According to the company's needs, the top priority was faster feature delivery, which means improving the team's agility. The second priority was reducing backend failures, achieved through better management of the microservices using Istio service mesh. The third priority was scalability. Although scaling the backend system was not urgent during this thesis, it is a task that will need attention in the future.

Table 1. The objectives weightage table of this thesis.

OBJECTIVES	Weightage
Agility. Faster feature delivery	5
Reliability. Less backend failure	3
Scalability. Easier to scale	2

Table 2 shows the priority of requirements for this thesis. The company required the author to use microservices architecture and GitHub Actions for CI/CD automation. These were essential to achieve the company's agility goals.

While using Istio service mesh and MicroK8s were not mandatory ("nice to have"), they were included for better reliability and scalability. Istio service mesh helps manage the microservices, and MicroK8s is used to host and run the containers.

Table 2. Priority table of requirements of this thesis.

Requirements	Must	Nice to have	More in the future
Use microservices architecture	Yes		
Use GitHub Actions for CI/CD	Yes		
Use Istio service mesh		Yes	
Use MicroK8s		Yes	
Decompose the rest of the application			Yes

As the two tables above show, there are three main goals in this thesis: agility, reliability, and scalability. From the customers' perspective, these goals translate to faster feature delivery, fewer server errors, and availability during traffic spikes.

Figure 1 shows the four main use cases of the mind mapping web application from the users' perspective.

1. **User Authentication**: This includes logging in, registering when an account does not exist. It requires third party authentication providers to verify the identity of users.
2. **Managing Diagrams**: Users can create, edit their diagrams, and save their diagrams to the cloud. This requires the user to be authenticated.
3. **Diagram Templates**: Users can use templates to create diagrams. A new diagram will be created if users use a template.
4. **Image and Icon Materials**: Users can search for, view, and use various image and icon materials in their diagrams. Users can not use materials without creating a diagram.

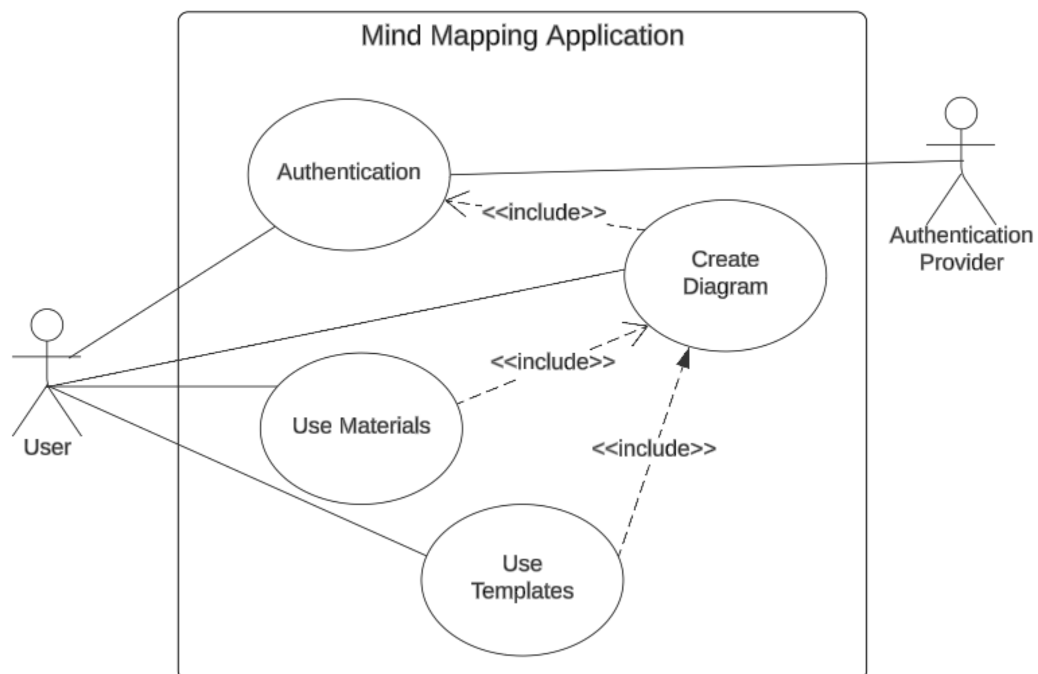

Figure 1. Use cases diagram of the mind mapping application of this thesis.

Figure 2 demonstrates the sequence diagram of the four use cases of the application. Users have two ways of creating diagrams: creating diagrams directly or using a template to create a diagram. The operation of creating a diagram always requires authentication from authentication providers. Once the diagram is created, users can use materials into that diagram.

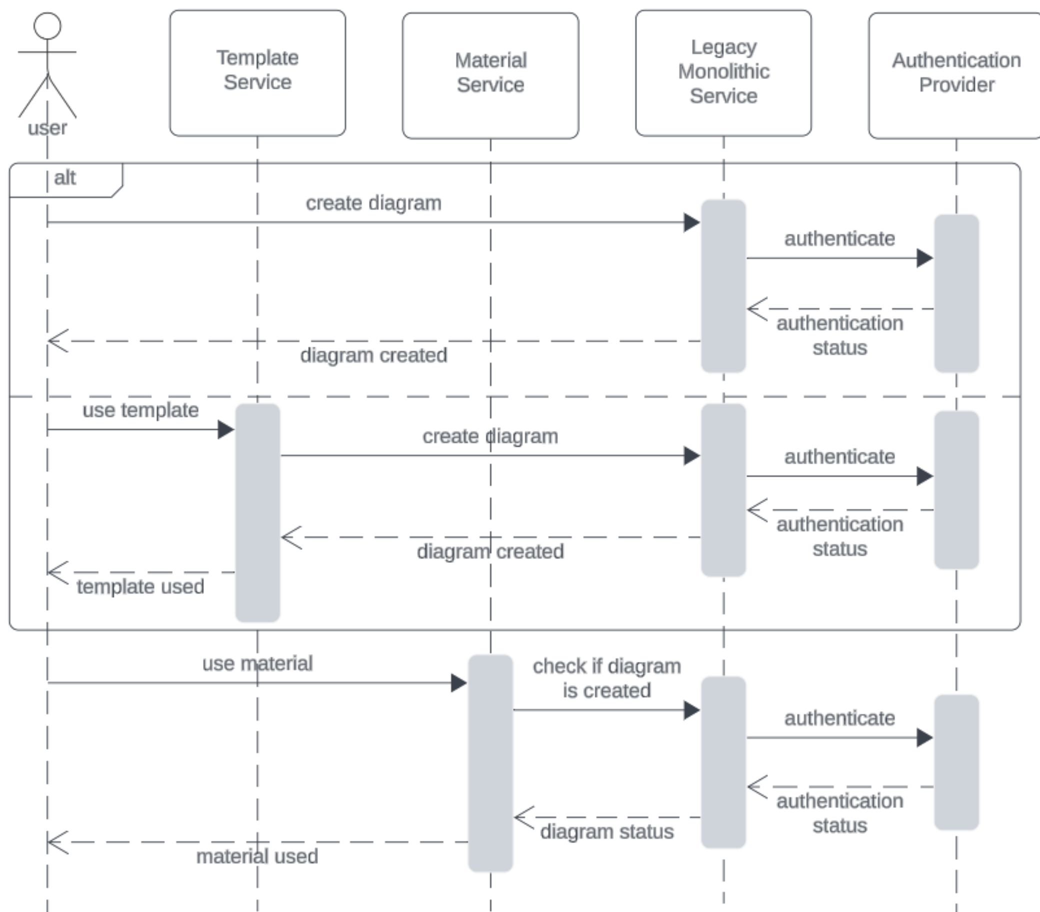

Figure 2. Sequence diagram of the mind mapping web application.

As shown in Figure 2, when users create diagrams directly, they initiate a diagram creation request to the legacy monolithic service. The legacy monolithic service then sends a request to the authentication provider to check the status of authentication and returns the diagram creation result to the users. If users choose to use templates to create a diagram, they first need to send a request to the template service. The template service then asks the legacy monolithic service to create a

new diagram, which will also trigger the authentication process. Once the diagram is created, users send a request to the material service to use a material for the diagram. The material service then communicates with the legacy monolithic service to check if the diagram the user intends to use material for has been created. This operation will also trigger the authentication process. Finally, the legacy monolithic service returns the diagram status to the material service, allowing the material service to use the material for that diagram and return the result of using the material to the user.

Figure 3 illustrates the CI/CD workflows implemented in this thesis. When developers push code changes to the main branch of the GitHub repository, GitHub Actions pipelines are triggered. All the unit tests would be running inside the GitHub Actions Runner environment. If all unit tests pass, the application is built into a Docker image, pushed to Docker Hub, and get the returned image URL.

Then, GitHub Actions Runner will move to the next workflow, deploying the latest version of the application (using the image URL) onto the MicroK8s cluster on the server. After deployment request to MicroK8s is sent, the GitHub Actions Runner will be destroyed. Once the MicroK8s deployment task is triggered, it will try to pull the new image from Docker Hub and run the image inside MicroK8s.

As a result, the entire process of testing, building, and releasing is done automatically.

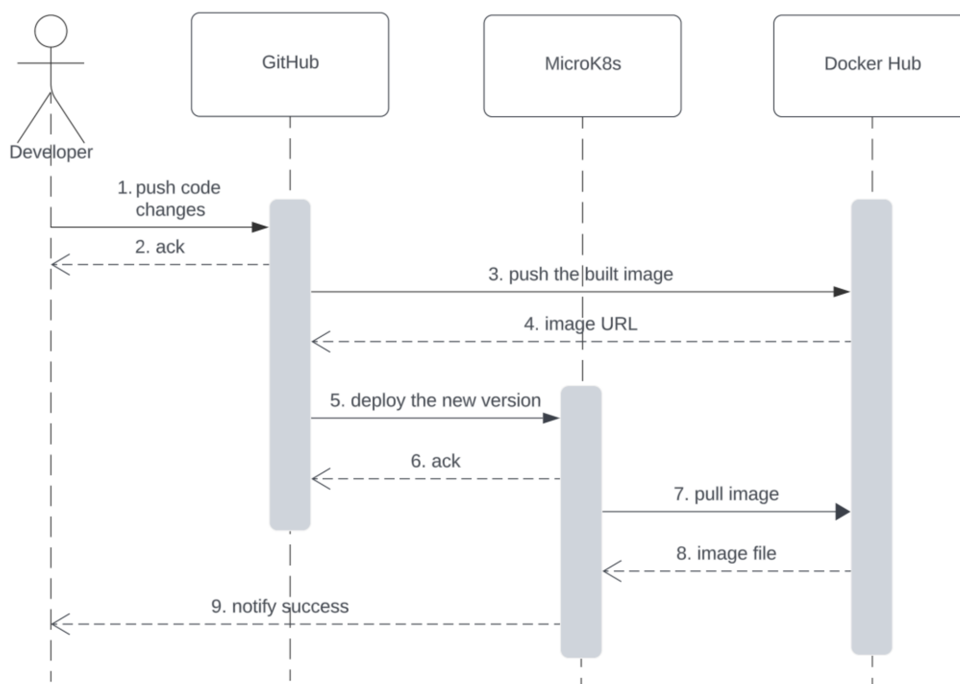

Figure 3. Sequence diagram of the CI/CD workflows of the application.

4 DATABASE DESIGN

This section describes the class diagram of the application and the PostgreSQL database schema design of the two new tables created for the two new microservices: “Template-API” and “Material-API”.

Figure 4 is a class diagram that describes the details of the four main classes in the application: user, diagram, template, and material. Each class has its own necessary properties and methods. There is no inheritance relationship between them.

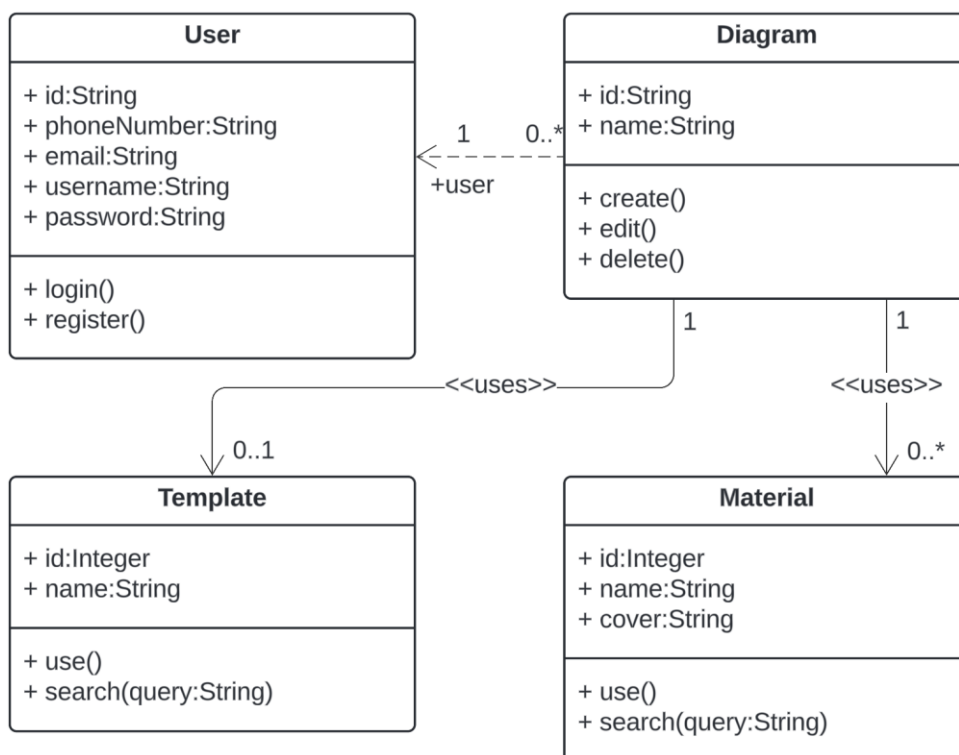

Figure 4. Class diagram of the application.

The user class must have properties like id, phone number, email address, username and password, methods like login operation and registration.

The diagram class must have properties like id, diagram name, the user id of its creator and the URL of the location where the actual diagram data is stored, and methods like create, edit, and delete. The diagram class relies on user class. If one of the users is deleted, all his diagrams would also be removed. A diagram would not exist apart from the user account. One user can have many diagrams.

The template class must have properties like id, template name and the URL of the location where the actual template data is stored, methods like use and search. Templates exist independently. Users can use templates in their diagrams. One diagram can use zero to one template.

The material class must have properties like id, name of the material, the URL of the location where the actual material data is stored and the cover URL of the material, and methods like use and search. Materials exist independently. Users can use materials in their diagrams. One diagram can use multiple materials.

Code snippet 1 is the table creation Structure Query Language (SQL) schema for "Template-API" service. It defines a table named "templates" which has an auto-incremental field "id" as its primary key, an URL field to locate the actual data stored in object storage, a title field to display in the browser, a cover field to locate the cover image path in object storage, a search vector field for searching functionality and a use count field to record the times customers used of this template. In the end, this code snippet created an index for search vector field to make the search operation in this table much faster.

```
create table templates(
	id serial not null primary key,
	url text not null,

	title text,
	cover text not null,
	search_vector tsvector,

	use_count int
);
create index on templates using GIN (search_vector);
```

Code Snippet 1. Schema creation SQL of the template table in Template-API service.

Code snippet 2 is the table creation SQL schema for “Material-API” service. It defines a table named “materials” which has an auto-incremental field “id” as its primary key, a cover field to locate the cover image path in object storage, a width field to store the width of the actual image, a height field to store the height of the actual image, a title field to display the title of the material to users, a search vector field for searching functionality, a URL field to store the path of the actual image file stored in object storage, a size field to store the actual image file size and a use count field to record the times of usage of this material. In the end, this code snippet created an index for search vector field to make the search operation in this table much faster.

```
create table materials(
	id bigserial not null primary key,

	cover text not null default '',
	width smallint not null default 0,
	height smallint not null default 0,
	title text,
	search_vector tsvector,

	url text not null default '',
	size bigint not null default 0,
	use_count smallint
);
create index on materials using GIN (search_vector);
```

Code Snippet 2. Schema creation SQL of the material table in Material-API service.

5 IMPLEMENTATION

This section explains the details of the application implementation, including the code structures of the main microservices, example code snippets for interservice communication, the configuration of MicroK8s, the deployment of Istio and Kiali, routing rules settings for Ingress in MicroK8s, and the GitHub Actions setup for CI/CD pipelines.

In this thesis, two new microservices were decomposed from a legacy monolithic application. The backend system now consists of the legacy service and two newly created microservices: "Template-API" and "Material-API."

A MicroK8s container orchestration environment was deployed, and Istio service mesh was integrated into the MicroK8s cluster, with Kiali enabled for monitoring the service mesh. Three Ingress routing rules were configured for load balancing purposes.

The GitHub Actions workflows were configured for CI/CD pipelines, and a Docker file was created to build the Docker images for the microservices.

5.1 Code Structure

Code snippet 3 shows the main structure of the "Template-API" service. First, it opens a connection to the PostgreSQL database using the URL configured in the container's environment variable. Then, it creates an API server and adds the necessary routing rules for various operations, including querying, getting, searching, using, and collecting templates. Finally, it prints information to the console and launches the API server to listen for incoming requests.

```
dbc, e := sql.Open("postgres", os.Getenv(ENV_POSTGRES))
if e != nil {
	log.Panic(e)
	return
}
defer dbc.Close()
server = apiserver.NewApiServer()
server.HandleFunc("/template/home", home)
server.HandleFunc("/template/query", query)
server.HandleFunc("/template/get", getTemplate)
server.HandleFunc("/template/search", search)
server.HandleFunc("/template/use-template", use)
server.HandleFunc("/template/collect", collect)

println("started at http://localhost:80")
e = http.ListenAndServe(":80", server)
if e != nil {
	log.Panic(e)
	return
}
```

Code Snippet 3. Code structure of Template-API service.

Code snippet 4 is the main code structure of the “Material-API” service. It firstly opens a connection to the PostgreSQL database with the URL configured in the environment variable of the container. Then it creates an API server, then adds necessary routing rules to the API server, including use, query, search, and view operations. Then it prints out information to the console and launches the API server to listen for incoming requests.

```
dbc, e := sql.Open("postgres", os.Getenv(ENV_POSTGRES))
if e != nil {
	log.Panic(e)
	return
}
defer dbc.Close()
server = apiserver.NewApiServer()
server.HandleFunc("/material/use", useMaterial)
server.HandleFunc("/material/query", queryMaterial)
server.HandleFunc("/material/search", search)
server.HandleFunc("/material/view", view)

println("started at http://localhost:80")
e = http.ListenAndServe(":80", server)
if e != nil {
	log.Panic(e)
	return
}
```

Code Snippet 4. Code structure of Material-API service.

5.2 MicroK8s

In this thesis, MicroK8s was installed in an Ubuntu virtual machine on the cloud via SHELL command in code snippet 5.

```
# Install MicroK8s on Ubuntu
sudo snap install microk8s –classic
```

Code Snippet 5. Shell script of installing MicroK8s on Ubuntu.

Appendix 1 is the application deployment configuration of “Template-API” in MicroK8s cluster. It consists of a Pod resource and a Service resource. The Pod resource hosts the actual application container. Its container configuration includes the URL of the image, the configuration of environment variables, container port. The Service resource exposes the Pod for outside resources to reach via network connection. Its selector is the label of the Pod resource to link the Services resource to the Pod resource. And the ports configuration stays consistent with the Pod resource.

Appendix 2 is the application deployment configuration of “Material-API” in MicroK8s cluster. It also consists of a Pod resource and a Service resource. The Pod resource hosts the actual application container. Its container configuration includes the URL of the image, the configuration of environment variables, container port. The Service resource exposes the Pod for outside resources to reach via network connection. Its selector is the label of the Pod resource to link the Services resource to the Pod resource. The ports configuration also stays consistent with the Pod resource.

5.3 Istio Service Mesh

Code snippet 6 is the Shell command for enabling Istio service mesh in MicroK8s cluster. It first enabled community add-ons, then enabled Istio. In the end use “kubectl” command to add sidecar injector for all the microservices in the cluster so that Istio can proxy and manage the traffic between services.

```
# Enable Istio on MicroK8s
microk8s enable community
microk8s enable istio

# Enable the Istio sidecar injector
kubectl label ns default istio-injection=enable
```

Code Snippet 6. Shell script of enabling Istio service mesh in MicroK8s.

Appendix 3 contains a code snippet demonstrating interservice communication between "Template-API" and the legacy service. There are two functions in the code snippet: "Put Collection" and "Delete Collection."

1. “Put Collection” Function: This function is triggered when a user clicks the collect button on the template page to collect a template. It first creates a key-value form with the necessary collection information and then posts the form to another service via an HTTP request. Once the HTTP call is completed, it returns the result of the operation.
2. “Delete Collection” Function: This function is called when a user tries to delete a template collection. It sends an HTTP DELETE request with the user ID and template ID concatenated into the URL to another service. Once the HTTP call is completed, it returns the result of the operation.

5.4 Load Balancing

Code snippet 7 demonstrates the routing rules used in this thesis. It uses prefixes to differentiate the target microservice for the HTTP request. The main host is the API gateway for all requests. It is decomposed into three different services with three routing patterns. Two of them, "/material" and "/template", are the two newly created microservices that are decomposed out of the legacy monolithic application. The rest of the API goes to the legacy monolith service.

```
kind: Ingress
spec:
  rules:
  - host: api.mindyushu.com
    http:
      paths:
      - pathType: Prefix
        path: "/template"
        backend:
          service:
            name: template-svc
            port:
              number: 80
      - pathType: Prefix
        path: "/material"
        backend:
          service:
            name: material-svc
            port:
              number: 80
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: baseapi-svc
            port:
              number: 80
```

Code Snippet 7. Load balancing configuration of the microservices application.

5.5 GitHub Actions

Code snippet 8 is a Docker file of building the Docker image for the application in this thesis. First, it used the Go official Docker image as a builder stage to run unit tests and compile the application to build the binary file. Then it copied that binary from the last stage to an alpine runtime which is a lightweight Linux distribution to host the binary application. This Docker file helps build a small, self-contained, independently deployable Docker image and ready for production.

```
# Dockerfile for building Go application
FROM golang as builder-stage
WORKDIR /app
ADD . .
RUN go test –v ./...
RUN CGO_ENABLED=0 go build –o main –trimpath –ldflags=”-s –w" .

FROM alpine:latest
WORKDIR /app
COPY –from=builder-stage /app/main /app/main
EXPOSE 8080
CMD [“/app/main”]
```

Code Snippet 8. Configuration of building Docker image for the application.

Appendix 4 shows the workflow configuration of GitHub Actions CI/CD. This file is executed on GitHub Actions' virtual machine when certain event happened: when developers pushed code changes to main branch, when developers added a new tag to the repository with format like "v1.2.3", when developers created new pull request targeting the main branch of the repository. These conditions are configured in the beginning of the workflow configuration file. Then, this file defines the running environment of the workflows which is Ubuntu operating system in this case and set up some permission to it and then do the actual work. This file consists of three jobs: test, build and deploy.

The test job is for unit testing of the code. It uses go command to run all the unit tests code in the project folder recursively. After all unit tests passed, this job succeeds.

The build job is for building the Docker image and push it to Docker Hub. First, it checks out the repository of the project and sets up Docker building tool environment. Then, it logs into Docker Hub with credentials pre-defined in GitHub variables page and runs the building and pushing operation for the project using Docker.

In the end, the deploy phase would deploy application. First, it loads the KUBE CONFIG information of the target MicroK8s cluster to the local file system and uses the configuration context. Finally, it runs the “kubectl apply” command to deploy the new version of the application to the MicroK8s cluster.

6 TESTING

The test cases performed were

1. Test GitHub Actions CI/CD pipelines.
2. Test the service mesh monitoring of all microservices.
3. Test the memory consumption of the MicroK8s cluster.

Figure 5 is the successful page of GitHub Actions CI/CD. It has three Jobs in the workflows: test, build and deploy. As the figure shows, these three jobs ran successfully in order.

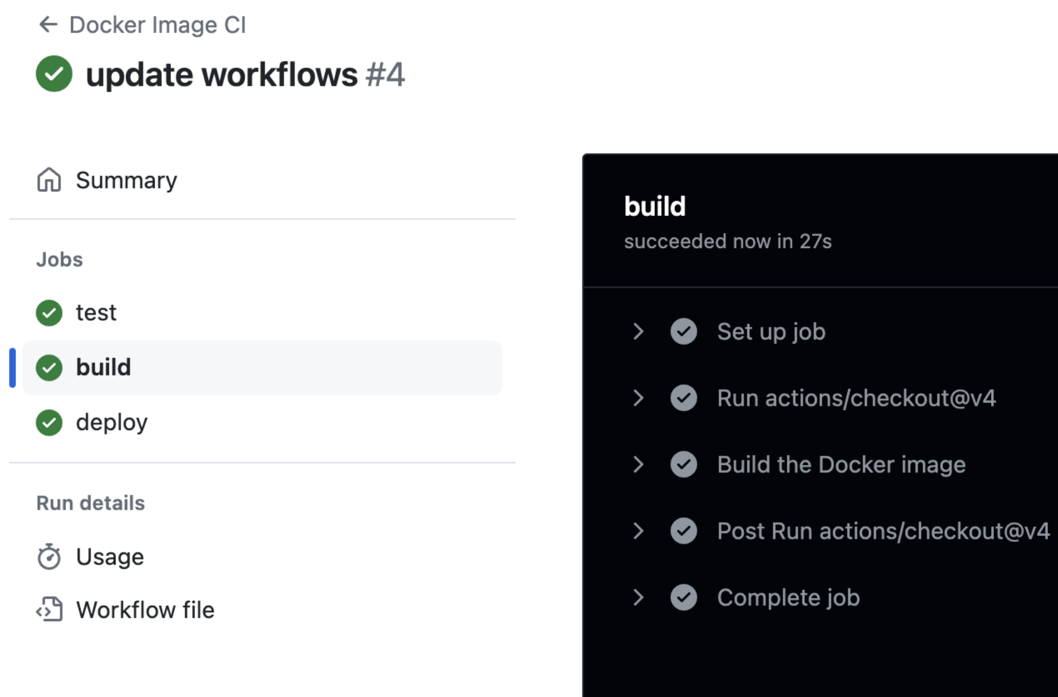

Figure 5. The result of GitHub Actions workflows. (Screenshot of Chrome browser)

Code snippet 9 is the result output of unit tests of the application. It listed all the modules that have unit test code at the beginning, and the time elapsed for running the unit tests, and then printed the test coverage of the modules in the output. This indicates that the new version of the application had passed all the unit tests which statistically proved that compatibility of the new version of the application.

```
ok common/const/languages 0.148s coverage: 33.3% of statements
ok common/const/ossprefix 0.166s coverage: 11.3% of statements
ok common/const/regions 0.273s coverage: 50.0% of statements
ok common/toolkit/audiox 0.173s coverage: 35.7% of statements
ok common/toolkit/cryptox 2.155s coverage: 28.4% of statements
ok common/toolkit/imagex 0.151s coverage: 8.5% of statements
ok common/toolkit/memberx 0.271s coverage: 4.2% of statements
ok common/toolkit/numx 0.400s coverage: 44.0% of statements
ok common/toolkit/sdkx 1.340s coverage: 13.9% of statements
ok common/toolkit/sdkx/ossx 0.371s coverage: 6.0% of statements
ok common/toolkit/strx 0.321s coverage: 22.2% of statements
ok common/toolkit/timex 0.451s coverage: 12.5% of statements
ok customer/service/member 0.263s coverage: 0.2% of statements
```

Code Snippet 9. The terminal output of unit test result of the application.

Figure 6 is a Chrome browser screenshot of service mesh monitoring graph. It was a test of traffic tracking and monitoring for the Istio service mesh. It recorded all the requests that flow inside the mesh. The traffic flows went with the direction of arrow lines in the graph. The graph also shows the network dependencies between microservices in the mesh. For example, the "Base-API" service relies on a Redis database. The two newly created microservices, "Template-API" and "Material-API", rely on a PostgreSQL instance.

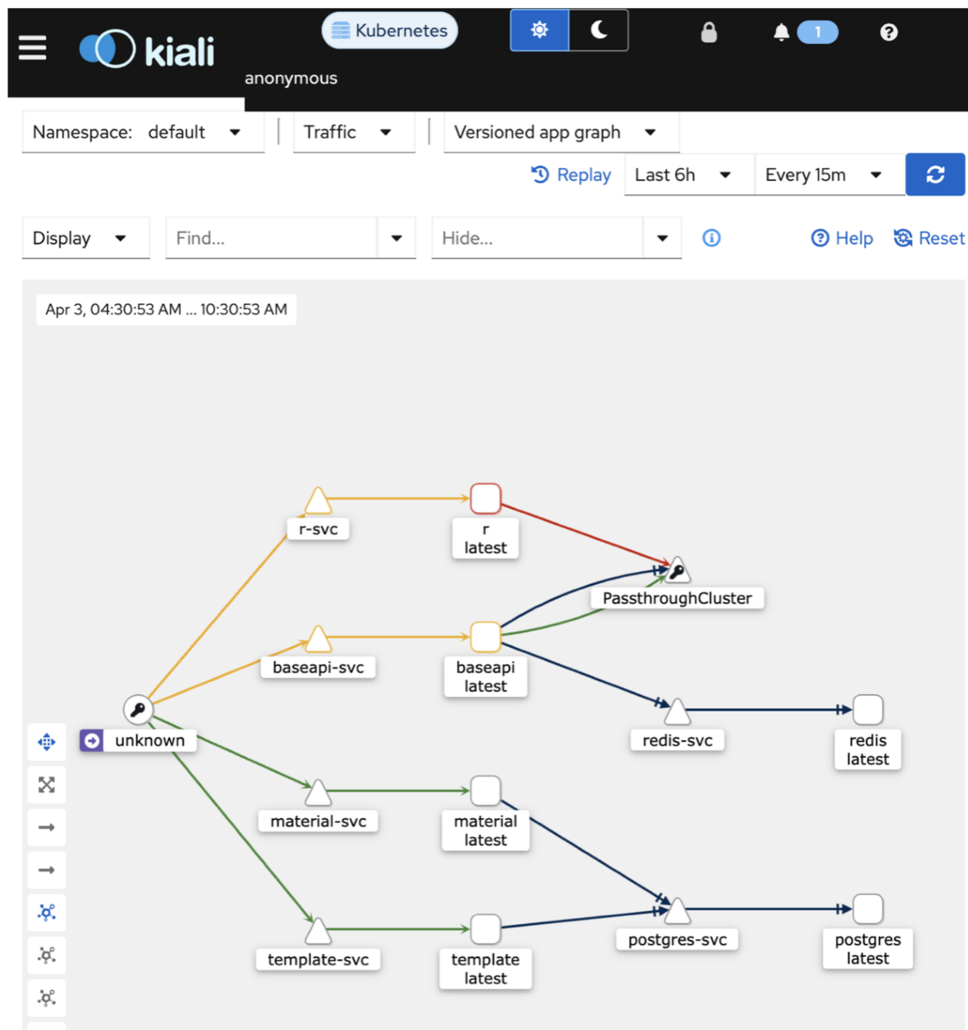

Figure 6. Traffic tracking result of the service mesh. (Screenshot of Chrome browser)

Figure 7 is a screenshot of terminal which demonstrates the memory consumption of the Ubuntu virtual machine used in this project. The application is running on an Ubuntu server with 8 Gigabytes of RAM, 20 Gigabytes of block storage and 8 virtual CPUs, which is bought from Alibaba Cloud. As the figure shows, after the MicroK8s cluster is deployed and all the components along with the application running inside the cluster, the total consumption of the computer memory is about 4.2 Gigabytes.

```
[asd@asd-pc:~$ free -h
              total        used        free      shared  buff/cache   available
Mem:          7.6Gi       4.2Gi       284Mi       126Mi       3.5Gi       3.3Gi
```

Figure 7. Memory consumption of the host Linux server. (Screenshot of SSH in terminal)

Based on the testing results above, the memory consumption of the cluster looks a little bit high. It can be slightly reduced by removing unnecessary components used in the Istio service mesh. It might need a thorough examination of the MicroK8s cluster to find out the problems.

7 SUMMARY

Table 3 lists all achievements of the requirements described in the beginning of this thesis. As the table shows, in this thesis, a legacy monolithic application was migrated into microservices architecture partially (Decomposed two microservices, “Template-API” and “Material-API”, out of the old application). GitHub Actions was also integrated to implement the DevOps practices of CI/CD, and Istio service mesh was used to manage traffic of the microservices group. Then, the application was deployed onto a container orchestration environment called MicroK8s. What has not been achieved was decomposing the rest of the application which was too large to do due to time limitations of this thesis.

Table 3. Achievement table of requirements in this thesis.

Requirements	Achieved
Use microservices architecture	Yes
Use GitHub Actions CI/CD	Yes
Use Istio service mesh	Yes
Use MicroK8s	Yes
Decompose the rest of the application	No

The initial goal of this thesis was to improve the speed of feature delivery, scalability, and reliability of the backend. Based on the achievements table, all three objectives were achieved relatively. The development team is now delivering new features much faster than before. Developers can locate and track the issues that happened in the backend easily with the help of service mesh which makes the

system more reliable. The operation of scaling the backend when spike access hap-
pened is simpler than before thanks to the use of MicroK8s.

8 CONCLUSIONS

In this project, most important goals were achieved. The three main objectives of this thesis: agility, reliability and scalability of the application were noticeably improved by the author's work.

Many tasks were completed in this thesis. The backend application was successfully restructured using a microservices architecture, and the databases were redesigned. GitHub Actions CI/CD was integrated to automate testing, building, and deploying the application. MicroK8s was used for hosting the containerized application, making the system ready to scale. Istio service mesh was enabled for better management of all microservices in the MicroK8s cluster. As a result, the application now works for both users and developers.

From the users' perspective, new features are now released more frequently. Users are receiving bug fixes and updates faster, which significantly improves their experience. They are also encountering fewer server errors because issues are caught before they noticed.

From the developers' perspective, they no longer have to deploy the application manually (compiling the application on their laptops and uploading the binary file to the server). The efficiency of the development team has significantly improved. Developers can now focus more on creative and valuable work instead of wasting time on tedious processes. They can also update the application in parts without worrying about breaking the entire system, thanks to the microservices architecture.

The most challenging part of this thesis was ensuring compatibility during the decomposition of the application. The two newly created microservices had to behave the same as the old application so that users wouldn't notice any differences during the migration process.

The future development based on this thesis involves migrating the rest of the monolithic application to a microservices architecture. Due to time constraints, this thesis only decomposed two microservices from the legacy application. The remaining parts of the application need to be migrated in the future by following the same steps outlined in this thesis: code migration, database migration, and designing interservice communication.

REFERENCES

Amazon Web Services. (n.d.) What are Microservices? Retrieved 2024-01-26 from https://aws.amazon.com/microservices/

Amazon Web Services. (n.d.) What is DevOps? Retrieved 2024-01-26 from https://aws.amazon.com/devops/what-is-devops/

Andrawos, M., & Helmich, M. (2017). Cloud Native Programming with Golang: Develop microservice-based high performance web apps for the cloud with Go. Packt Publishing Ltd.

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Migrating to cloud-native architectures using microservices: an experience report. In Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2015, Taormina, Italy, September 15-17, 2015, Revised Selected Papers 4 Springer International Publishing.

Bass, L., Weber, I., & Zhu, L. (2015). DevOps: A software architect's perspective. Addison-Wesley Professional.

Böhm, S., & Wirtz, G. (2021, March). Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes. In ZEUS.

Combe, T., Martin, A., & Di Pietro, R. (2016). To docker or not to docker: A security perspective. IEEE Cloud Computing.

Decan, A., Mens, T., Mazrae, P. R., & Golzadeh, M. (2022, October). On the use of GitHub Actions in software development repositories. In 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE.

Dilley, N., & Lange, J. (2019, February). An empirical study of messaging passing concurrency in Go projects. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE.

Douglas, K., & Douglas, S. (2003). PostgreSQL: a comprehensive guide to building, programming, and administering PostgreSQL databases. SAMS publishing.

Drake, J. D., & Worsley, J. C. (2002). Practical PostgreSQL. O'Reilly Media, Inc.

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. IEEE software, 33(3).

GitHub. (n.d.) Understanding GitHub Actions. Retrieved 2024-01-26 from https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

Httermann, M. (2012). DevOps for developers. Apress.

Istio Team. (n.d.) The Istio Service Mesh introduction. Retrieved 2024-01-26 from https://istio.io/latest/about/service-mesh/

Koziolek, H., & Eskandani, N. (2023, April). Lightweight Kubernetes distributions: a performance comparison of MicroK8s, k3s, k0s, and MicroShift. In Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering.

Luksa, M. (2017). Kubernetes in action. Simon and Schuster.

McGrath, M. (2020). GO Programming in easy steps: Discover Google's Go language (Golang). In Easy Steps Limited.

MicroK8s. (n.d.) What is MicroK8s? Retrieved 2024-01-26 from https://microk8s.io/

Microsoft Azure. (n.d.) Microservices architecture design. Retrieved 2024-01-26 from https://learn.microsoft.com/en-us/azure/architecture/microservices

Miell, I., & Sayers, A. (2019). Docker in practice. Simon and Schuster.

Mouat, A. (2015). Using Docker: Developing and deploying software with containers. O'Reilly Media, Inc.

Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2017). Cloud container technologies: a state-of-the-art review. IEEE Transactions on Cloud Computing.

Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2017). Cloud container technologies: a state-of-the-art review. IEEE Transactions on Cloud Computing.

Ponce, F., Márquez, G., & Astudillo, H. (2019, November). Migrating from monolithic architecture to microservices: A Rapid Review. In 2019 38th International Conference of the Chilean Computer Science Society (SCCC). IEEE.

Potdar, A. M., Narayan, D. G., Kengond, S., & Mulla, M. M. (2020). Performance evaluation of docker container and virtual machine. Procedia Computer Science.

Surianarayanan, C., Ganapathy, G., & Pethuru, R. (2019). Essentials of microservices architecture: Paradigms, applications, and techniques. Taylor & Francis.

Valenzuela, Toledo, P., & Bergel, A. (2022, March). Evolution of GitHub action workflows. In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE.

Zhu, X., She, G., Xue, B., Zhang, Y., Zhang, Y., Zou, X. K., ... & Mahajan, R. (2022). Dissecting service mesh overheads. arXiv preprint arXiv:2207.00592.

APPENDICES

APPENDIX 1

CODE SNIPPET OF APPLICATION CONFIGURATION OF TEMPLATE-API SERVICE

```
apiVersion: v1
kind: Pod
metadata:
  name: template-pod
  labels:
    app: template
spec:
  containers:
  - name: template
    image: ***/templateapi:latest
    env:
      - name: MODE
        value: release
      - name: REGION
        value: hk
      - name: POSTGRES
        value: ***
    ports:
    - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: template-svc
spec:
  selector:
    app: template
  ports:
  - port: 80
    targetPort: 80
```

APPENDIX 2

CODE SNIPPET OF APPLICATION CONFIGURATION OF MATERIAL-API SERVICE

```
apiVersion: v1
kind: Pod
metadata:
  name: material-pod
  labels:
    app: material
spec:
  containers:
  - name: material
    image: ***/materialapi:latest
    env:
      - name: MODE
        value: release
      - name: REGION
        value: hk
      - name: POSTGRES
        value: ***
      - name: MONGODB_SRC
        value: ***
    ports:
      - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: material-svc
spec:
  selector:
    app: material
  ports:
  - port: 80
    targetPort: 80
```

APPENDIX 3

CODE SNIPPET OF INTERSERVICE COMMUNICATION OF THE APPLICATION

```
func PutCollection(uid, cid, tid string, v db.Template, r *http.Request)
error {
	vs := make(url.Values)
	vs.Set("title", strx.GetTitle(v.Titles,
		strx.SupportedLanguage(r)))
	vs.Set("subtitle", vs.Get("title"))
	vs.Set("cover", v.Cover)
	res, e :=
	httpClient.PostForm(ACCOUNT_SVC+"/u/"+uid+"/col/"+cid+"/tmpl/"+ti
	d, vs)
	if e != nil {
		log.Println(e)
		return e
	}
	if res.StatusCode == 200 {
		return nil
	}
	defer res.Body.Close()
	b, e := io.ReadAll(res.Body)
	if e != nil {
		log.Println(e)
		return e
	}
	return fmt.Errorf("post collect failed:%s", string(b))
}

func DeleteCollection(uid, tid string) error {
	req, e := http.NewRequest(http.MethodDelete,
		ACCOUNT_SVC+"/u/"+uid+"/col/0/tmpl/"+tid, nil)
	if e != nil {
		log.Println(e)
		return e
	}
	res, e := httpClient.Do(req)
	if e != nil {
		log.Println(e)
		return e
	}
	if res.StatusCode == 200 {
		return nil
	}
	defer res.Body.Close()
	b, e := io.ReadAll(res.Body)
```

```
	if e != nil {
		log.Println(e)
		return e
	}
	return fmt.Errorf("delete collect failed:%s", string(b))
}
```

APPENDIX 4

CODE SNIPPET OF WORKFLOWS CONFIGURATION OF GITHUB ACTIONS

```
name: Docker
on:
  push:
    branches: [ "main" ]
    tags: [ 'v*.*.*' ]
  pull_request:
    branches: [ "main" ]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3
      - name: Unit test of the golang code
        run: go test -v ./...
  build:
    runs-on: ubuntu-latest
    needs: test
    permissions:
      contents: read
      packages: write
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3.0.0
      - name: Log into registry ${{ env.REGISTRY }}
        if: github.event_name != 'pull_request'
        uses: docker/login-action@v3.0.0
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
      - name: Build and push Docker image
        id: build-and-push
        uses: docker/build-push-action@v5.0.0
        with:
          context: .
          push: ${{ github.event_name != 'pull_request' }}
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
```

```
          cache-to: type=gha,mode=max
  deploy:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3
      - name: Load kube config file
        run: |
          mkdir ${HOME}/.kube
          echo ${{ secrets.KUBE_CONFIG }} | base64 --decode >
${HOME}/.kube/config
      - name: Use kube context and deploy
        run: kubectl config use-context mycontext && kubectl apply -f
k8s/
```