

Migrating from Monolithic Application to Microservices

Thi Tran

Bachelor's Thesis
Degree Programme in
Business Information
Technology
2020



Author(s) Thi (Trong) Tran	
Degree programme Business Information Technology	
Report/thesis title Migrating from Monolithic Application to Microservices	Number of pages and appendix pages 46 + 6
<p>The following work focuses on the concept of microservices and the adoption process from monolithic application to microservices.</p> <p>Microservices has been proven to be a preferable choice for enterprises to develop software. About 84% of organizations have embraced microservices architecture to accelerate innovation efforts and stay competitive. However, the migration process to microservices is not clear for organizations. 32% of organizations admit that difficulty integrating with monolithic legacy apps remains a challenge for their microservices adoption.</p> <p>This work shows how the migration process to microservices can be applied practically by implementing a case study on an eCommerce store. The work uses and describes in detail Strangler pattern as well as with the service domain as a framework for implementation. In the end, there will be an evaluation using a monitoring system form Postman to determine the availability and resilience of the system during the migration process.</p> <p>Results collected from the monitoring system are discussed and analyzed. They show that using Strangler pattern in migration process keeps the system availability over 97%, including system configuration and user errors. This result shows that Strangler pattern works well as a migration process method and allows organizations to have an incremental improvement over migration while changes are not likely noticed by customers. Although to ensure that the migration process succeeds, further research has to be conducted based on the complexity of the system.</p>	
Keywords Microservices, migration process, Strangler pattern, service domain.	

Table of contents

1	Introduction	1
1.1	Thesis goals and objectives	1
1.2	Scope of the thesis	2
1.3	Relevance of the results	2
2	Theoretical Introduction of Microservices	3
2.1	Microservice Architecture	3
2.2	Factors of Microservices	3
2.3	Benefits of Microservice	6
2.4	Challenges of Microservices	8
3	Technical Implementation Theory	14
3.1	Containerization	14
3.1.1	Containerization	14
3.1.2	Containerization and Microservices	14
3.2	Container Orchestration	15
3.2.1	Kubernetes	15
3.2.2	Kubernetes and Microservices	17
3.2.3	Services Communication in Kubernetes	20
4	Implementation Framework	23
4.1	Service Domain	23
4.2	Strangler Pattern	25
5	Implementation - Migration to Microservices	28
5.1	Project background	28
5.2	Implementation plan	29
5.3	Splitting monolithic application into microservices	31
5.4	Setting up a proxy system	34
5.5	Developing new services	35
5.6	Handling Connections	37
5.7	Conclusions	39
5.7.1	Reflections on monitoring data	39
5.7.2	Reflections on migration framework	40
6	Discussions	43
	References	45
	Appendices	48
	Appendix 1. Screenshot from Postman monitor	48
	Appendix 2. Screenshot of failed attempts' test result	50
	Appendix 3. List of abbreviations	53

1 Introduction

Microservices has been an advanced architecture pattern for years and been proven to be a preferable choice for enterprise to develop software. According to the 2020 Digital Innovation Benchmark, 84% of organizations have embraced microservices architecture to accelerate innovation efforts and stay competitive (Kong 2020). Success stories of companies such as Amazon, Google, Zalando or Spotify migrating to microservices (Kwiecień 2019) to stay competitive have inspired organizations to start adapting this modern architecture pattern.

Although the benefits of microservices over monolithic application are clear, organizations still struggle to migrate into microservice from their legacy monolithic architecture due to the complexity of the process. 32% of organizations admit that difficulty integrating with monolithic legacy apps remains a challenge for their microservices adoption (Kong 2020). This thesis is to approach this problem to make the transition simpler, along with maintaining system performance and avoiding unexpected downtime and risks during the migration process.

1.1 Thesis goals and objectives

The following work focuses on microservices, proposing approaches and best practices for companies to transform from monolithic architecture application to microservices architecture application. It provides an overview of microservices and its related concepts such as containerizations, container orchestration, cloud services providers. The thesis then reflects on existing good implementation from companies that successfully implemented microservices such as IBM, Google, etc.

The approach is then tested in a case study with codebase from backend of an eCommerce store that was written in monolithic architecture. The main expected outcome is successfully transforming backend code from monolithic to microservices using the approach proposed.

The goal of writing the thesis is to research and provide a concrete process for enterprises to migrate from Monolithic Application to Microservice Application. The process aims to make the transition step by step but still keep the services running on the server. The ideal outcome of the process is its ability to be applied to the current development of enterprises.

1.2 Scope of the thesis

The thesis only focuses on the process of transition into microservices in backend technologies, ranging from how to split microservices to developing microservices environment. Other concerns regarding microservices such as Health Monitor, API management, Service Security, Config Management, or splitting into microservices for frontend fall out of the scope of this work.

This work is also not intended to serve as the only-one-to-go option for users as the process depends exclusively on the situation of the system. Best practices and approaches chosen are based on the personal opinion of thesis writer. However, it can be considered as a suggestion and can be modified based on the needs of the organization.

1.3 Relevance of the results

Results derived from the work will be relevant for organizations that are interested in migrating their existing system to microservices. The example of approach and setting up the environment can be used as a template and adjust based on the company's needs and their system.

2 Theoretical Introduction of Microservices

In this section, we will discuss the theoretical background of microservices. We will elaborate on what it is and what advantages/challenges it brings.

2.1 Microservice Architecture

Microservices Architecture (will be referred to as microservices later on) is a system architecture for an application as a collection of services. Newman (2018, Chapter 1) defines microservices as small, autonomous services that work together. Different from monolithic architecture where one single application handles all business purposes and is responsible for all requests, microservices separate it into smaller applications that modeled around a given domain. These applications work independently for their domain and run on their container. However, if the requests need multiple services from different domains to finish the job, microservices can communicate with each other. Therefore, microservices can be referred to as "... a microservice architecture is based on multiple collaborating microservices" (Newman, Sam 2019).

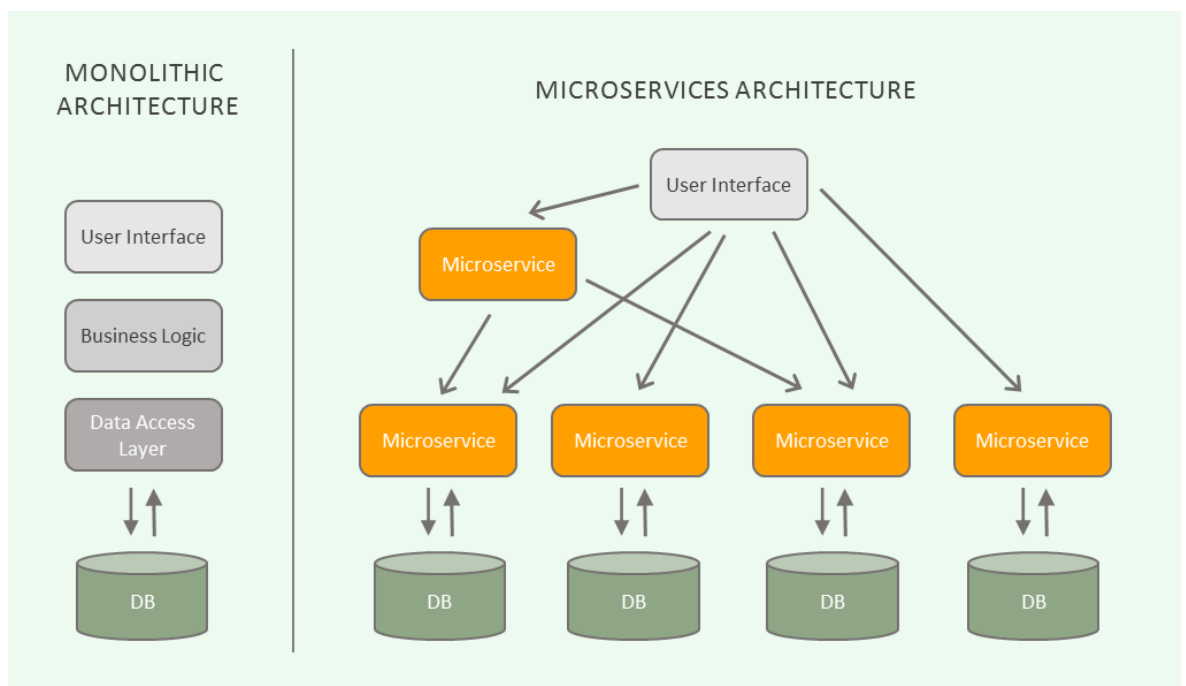


Figure 1. Monolithic Architecture and Microservices Architecture (adapted from Arsov 2017)

2.2 Factors of Microservices

Interdependence

Codebases grow every day as developers add new features. Over time, it is difficult to know where changes should be made to keep the code clean and not repeat itself.

Therefore, developers find it hard to fix bugs or implement them. In monolithic world, companies overcome this challenge by ensuring cohesiveness within the code by abstractions and modules (Newman 2018).

Microservices take a similar approach for each service: gathering codes under the umbrella of a domain where code lives for an “end-to-end slices of business functionality” (Newman 2019). However, microservices take a step further from modules as these services lay on top of an isolated machine, i.e. container (will explain later in the document), and model around a domain. As such, each service acts as a separate entity, having independent deployability.

For services in microservices, it is important to consider domain boundaries that service covers to maintain its interdependence. A rule of thumb is that service domain should not be too big or too small. If it is small, it will not cover enough functionality needed. Hence, one has to make changes across a process boundary for two separate services, which is against independent deployability of microservices. On the other hand, microservices should not be too big as it can return to monolithic architecture. The downside of this approach is that one finds it hard to make changes as it might affect other features in the service. In general, domain is considered to keep microservices size in good balance, so it has to be well-discussed to create services that are well covered and maintainable.

While each service has to separate into its environment, each service can communicate using an application programming interface (API). Each service has its internal Internet Protocol (IP) address within the network, which exposes APIs for other services to communicate back and forth. (Bruce & Pereira 2019.) Therefore, microservices can communicate with each other, and getting information from other services if needed. For example, two services Order and User can communicate via APIs. When Order needs more information about user details, it can send API requests to User service. Then, User service will return Order user details (often in JSON form). As such, each service is responsible for its functionalities, but still being able to collaborate to contribute to the overall system.

Owning their own data

Owning to its interdependency, microservices should not share database. If one service needs to access data held by another service, it should go and ask that service for the data. (Newman 2019). Werner Vogels, Amazon CTO, also refers to database separation as service orientation means encapsulating the data with the business logic that operates on the data, with the only access through a published service interface. No direct

database access is allowed from outside the service, and no data is sharing among the services. (AcmQueue 2020.)

By encapsulating databases within a service contributes, one can make changes without worrying about compatibility, adding velocity to development and deployment (Newman 2019). For example, if two services User and Order have a shared database and a developer wants to change schema of User table, he or she has to ensure that the change is compatible with Order service. Moreover, having a separate database ensures that each service can use the type of database that is best suited to its needs. In some circumstances, User service is better off using NoSQL database as user information is large sets of distributed data, whereas, Order service prefers to use relational database to keep track of orders. On the other hand, using a separate database, one can deploy service independently, taking away the hustle of trying to make two or three services working together.

Independent and dynamic team

Conway states that any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure (Newman 2018). In order to develop interdependent services, teams are required to become more agile - "more independent, autonomous teams, able to be responsible for more of the end-to-end delivery cycle than ever before" (Newman 2019). In fact, each team is independently working on one service while maintaining communication with other teams. The shift is from "traditionally grouping people in terms of their core competency" to "poly-skilled teams, to reduce hand-offs and silos" (Newman 2019). Figure 2 illustrates an example of team shifting within an organization.

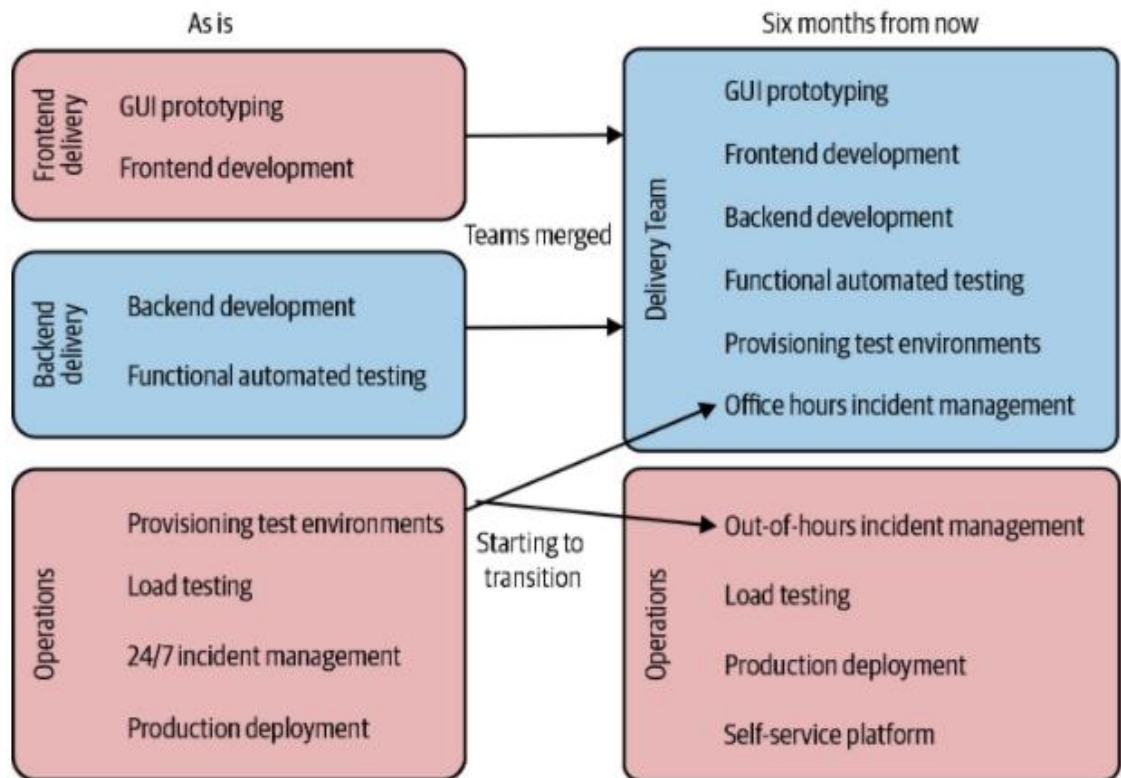


Figure 2. Reassign responsibilities in organizations (adapted from Newman 2019)

As each service and team is independent of others, each team can focus on building features without worrying about affecting codes of other teams, which helps teams to move fast. Werner Vogels, Amazon CTO, shares his experience in building Amazon as the giant, monolithic "bookstore" application and giant database that we used to power Amazon.com limited our speed and agility. Whenever we wanted to add a new feature or product for our customers, like video streaming, we had to edit and rewrite vast amounts of code on an application that we'd designed specifically for our first product—the bookstore. This was a long, unwieldy process requiring complicated coordination, and it limited our ability to innovate fast and at scale. (Vogels 2018.)

2.3 Benefits of Microservice

While some companies are still using monolithic architecture for its product due to simplicity of development, deployment and management, companies such as Google, Amazon, Netflix and Spotify have migrated into microservices because of its advantages over monolithic architecture.

According to 2020 Digital Innovation Benchmark, improvements to security (56 percent), increased development speed (55 percent), increased speed of integrating new technologies (53 percent), improved infrastructure flexibility (53 percent) and improved

collaboration across teams (46 percent) consistently mentioned as drivers of adoption. (Kong 2020.)

System resilience

Microservices improves system resilience. If one component of a system fails, but that failure doesn't cascade, you can isolate the problem and the rest of the system can carry on working (Newman 2019). On the other hand, as each microservice is an independent component communicating over APIs when there is a demand for workloads, the deployment team can decide to scale up some services (or one service) from the application instead of having to redeploy another instance of the application. Then, when the peak time is over, the development team can scale down these services.

Security

Microservice in its service independent nature provides a robust foundation for security. Antti Vähä-Sipilä, Principal Security Consultant of F-secure, has shared his view on this in a podcast as for data breach avoidance: that's actually a really interesting pattern because you will require all your users to go through a very well-specified API that's hopefully very robust and tested. And each microservice will also own their own data and they won't offer any sort of like backdoor access to the databases, for example, that somebody could use to dump the stuff. So they would have to do that through the API. So if you have like this sort of chain of microservices, one service calling another one, and when you even lock down the network connections between those, you end up with a pretty robust basic architecture. (Michael 2019.). Moreover, microservices make it harder for hackers. Antti depicts the issue as at least the attacks have to happen immediately ... I mean, you cannot just like put up house on somebody's server because the server is going away very soon. But it doesn't necessarily do much for like a traditional SQL injection. If you're vulnerable for that, it doesn't really help, if you have a kind of ephemeral node. (Michael 2019.). As such, migrating to microservices does not fully solve the security itself but rather depending on how the organization develops system in microservices. In another word, microservices provides a good foundation for security and it is up to organizations to utilize it.

Technology Heterogeneity

On the other hand, microservices opens more possibilities for development. As each microservice is independent of another, develop team can decide what language or framework they want to use. Therefore, it not only provides the team freedom to choose

technology that best fit for the task, and their preference, also helps prevent decisions made from the past affecting decisions in the future (In monolithic application, companies often choose one-fit-all solutions, which shows limitation for adapting new technology).

Another benefit of having independence services is that developers can escape from "dependency hell". In monolithic architecture, if a developer wants to change an old dependency, one is forced to change in codebase (sometimes in thousands of files) so that it all matches with the latest change. By keeping microservices code relatively small, developers can update dependency when needed to avoid any potential vulnerability.

Agile Development

Independence also shows benefits in deployment as each tech can build their own pipelines and develop the service independently at any time. Dave Hahn, a senior engineer in Netflix's cloud operations and reliability team, shares his experience in the company: "I don't have to assemble all of these pieces built by other people in order to have a singular deployment ... Any Netflix service team can deploy their service at any time. It requires no coordination, no scheduling, no crucible to get to production. (Macaulay 2018.). Moreover, microservice makes it easier and less risky to update the application as deployment only happens in one service, keeping other services safe during the time.

Independent deployability also contributes to improve development speed and shorten release time. Many organizations have found that by embracing fine-grained, microservice architectures, they can deliver software faster and embrace newer technologies. Microservices give us significantly more freedom to react and make different decisions, allowing us to respond faster to the inevitable change that impacts all of us. (Newman 2018.)

2.4 Challenges of Microservices

While building with microservices provides several benefits for organizations, it also comes with costs. Luckily, as the technology has been for years, Microservices is known as an architectural style with well-understood benefits and trade-offs. As microservice architecture grows over time, the number of instances running increases to hundreds or thousands. Therefore, shifting from managing one to three instances to thousands running simultaneously can be challenging. In another word, microservices brings many benefits for organizations, but it also comes with several problems itself. In this section, thesis writer will go through some most common challenges that organizations might encounter

when they migrate from monolithic to microservices while providing short recommendations to resolve the problem.

Continuous Integration

Continuous Integration (CI) plays an important role in implementing microservices efficiently. The core goal of CI is to keep everyone's code in sync with each other by checking newly added code can properly integrate with existing code. As microservices grow and teams start to contribute, CI helps to reduce the manual working of QA engineers (and also developers!) to look at the large amount of code submitted from teams in different services (Newman 2018). Before migrating to microservices, organizations should have already implemented their CI pipeline system and ready to reuse the experience of building that pipeline to build CI system for microservices. Although, testing in microservices can be slightly different from monolithic testing where it requires at least testing between services.

One common anti-pattern from migrating CI from monolithic into microservices is trying to have one single repository and one CI build for multiple services(Newman 2018). On the surface, it conceptually makes simpler build and easier to track errors later. However, this approach comes with significant downsides. For example, if the developer only makes a one-line change to a single service, then all other services will automatically get built and verified in CI pipelines. This takes more time and resources needed and slows down development time. Moreover, it gets even more troublesome trying to decide which artifacts to build just based on the commit message (Newman 2019). One alternative of this practice is to still keep one big repository, but having CI pipelines for different services. This achieves both ease to check-in/check-out of developers and waste resources of CI builds. However, developers have to be cautious in writing their code as they can easily slip into making changes multiple services at once, which should be avoided. The ultimate solution for this is to have CI build per microservice, i.e. each microservices has its own repository and CI pipelines (Newman 2018). Hence, new code will on be tested in the service it lives on and at the same time making it difficult to one change for multiple services. On the other hand, this method also supports team alignment in microservices as each team owns the service, hence owns the repository and its pipeline.

Testing

In monolithic world, we are used to having testing around an individual service. However, since microservices adopts splits these into multiple services and each communicates

over the network, tests to cover for not only service but also testing in context with other services around it while maintaining factors of microservices to allow teams to achieve the benefits of it.

With automated testing, microservices generally has three main types of testing: unit tests, service tests, UI tests (end-to-end test). Cohn (2010) depicts these types of Test Pyramid in figure 5.

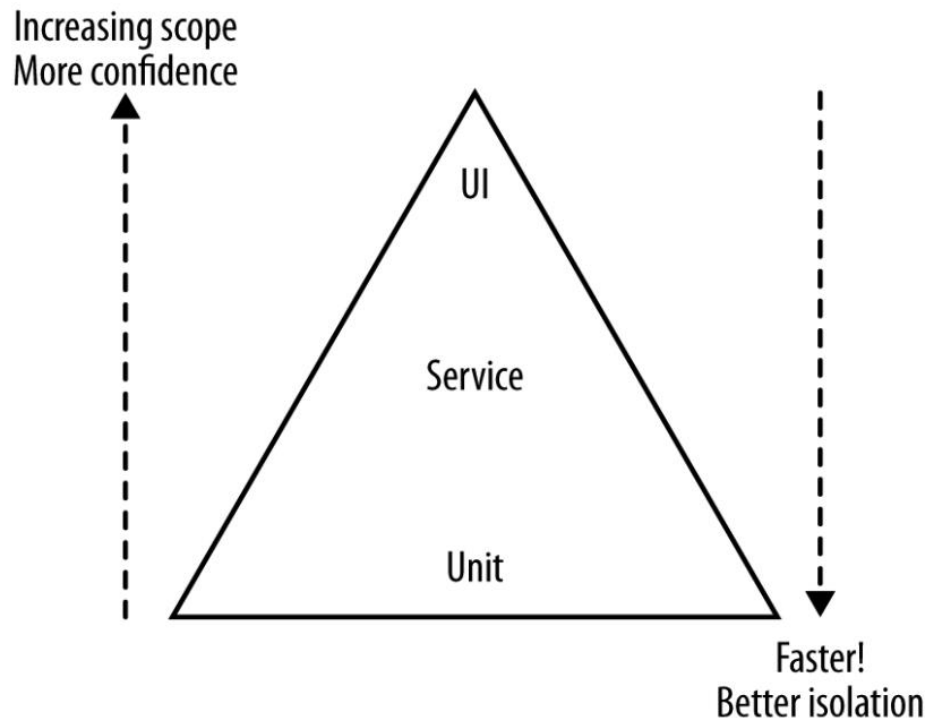


Figure 3. Mike Cohn's Test Pyramid (adapted from Cohn 2010)

However, these tests require lots of resources and sometimes can be very tricky in microservices environment. For example, with Order and User service, when Order service goes through CI pipelines, it is hard to define which version of User service is needed to run tests with. If developers choose to have in the production version, there might a version of User service is in the queue waiting to go live and testing has to run again once the new version of User is deployed. Hence, end-to-end tests could diminish the independent deployability benefit of microservices (Newman 2018). Moreover, the more components involve in the test, the more likely some unintentional failure such as race condition, timeout, etc. could happen. It makes the test flaky, i.e. developers do not know where actual errors or bugs lie.

There are also other alternatives model for testing in microservices. Schaffer (2018), ex-Engineering Manager of Spotify, has proposed Microservices Testing Honeycomb model (although the names of testing types are different) where emphasizes service tests and

lessens on end-to-end tests. For unit tests, André Schaffer refers them for parts of the code that are naturally isolated and have an internal complexity on their own.

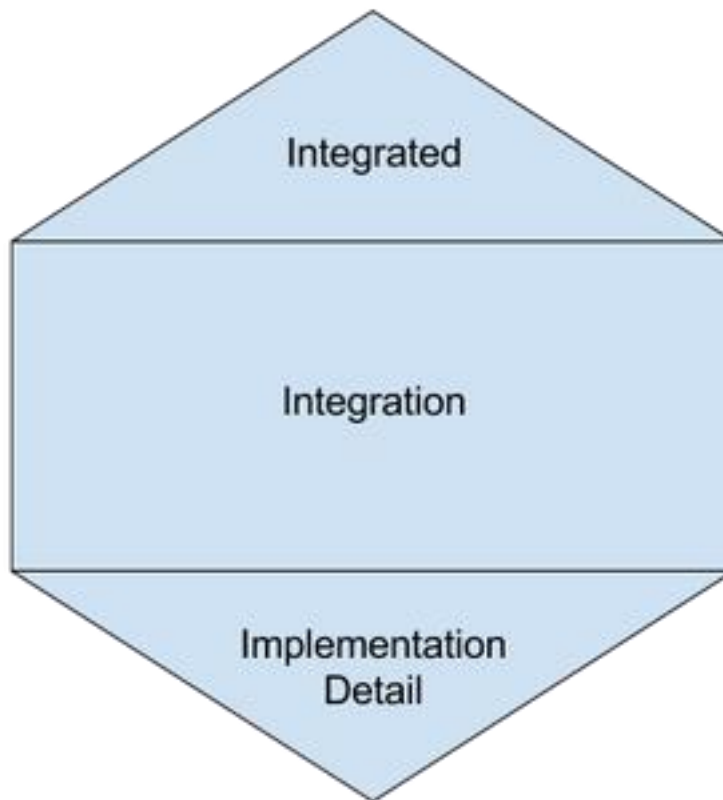


Figure 4. Microservices Testing Honeycomb (adapted from Schaffer 2018)

Debugging

Debugging in microservices is generally more challenging than in monolithic application. Some requests require to bounce between different services, which makes it harder to find the root of the problem.

It requires a holistic approach to the challenge. First of all, the system should be designed with highly cohesive and loosely coupled services (will be discussed later on). If each service is independent and focuses on its boundary, most of the bugs will be encapsulated within one specific service. Secondly, microservices requires to monitor and logging system. If you need to debug a distributed system, the monitoring and tracking capabilities act as a flight recorder. You should have all that happens stored there so you can later investigate every single event to make sense of what happened in a multitude of systems. (Bruce & Pereira 2019.) Lastly, communication between services should support loosely coupled services. As services become dependent on each other, it is harder to trace the bug later on. As a solution, choreography communication will be discussed later on.

Logging

Logging (or log analytics and management) is undeniably an essential element in an information system because it gives insight into how the system is doing and what happened to it in case something goes wrong. When migrating into microservices, the operators team has to manage and collect logs from not only different services but also in each host that the services are running on (Newman 2018). Moreover, as microservices often communicate with each other to fulfill requests from users, it gets complicated to trace back an error that is hidden behind the chain of multiple hosts of different services.

In monolithic architecture, stdout logging is often used as a solution due to its simplicity and effectiveness. Generally, stdout works as the log libraries push application's logs to monitoring tools for every defined period or every group of log collected. As logs from the application already stayed in structure, there is no need to order them in monitoring tools. (Newman 2018.)

In microservices, this is not the best solution because of stdout's nature as a library. Microservices allow developers to independently update service and freedom to choose the preferred language, stdout limits its ability by having to concerns about its library updates and supports for the selected language.

Instead, a common solution is to export a log file and having log collectors to gather logs from different instances into one place at specific time intervals (Özgür 2019). Fortunately, with microservices running on the container and orchestrated by Kubernetes (will be explained later), Logstash or Fluentd with their respective implementation Filebeat or Fluentbit is an open-source project for this purpose. After that, logs can be sent to Elasticsearch to store and visualized using another open-source project: Kibana. The following process can be explained in figure 7.

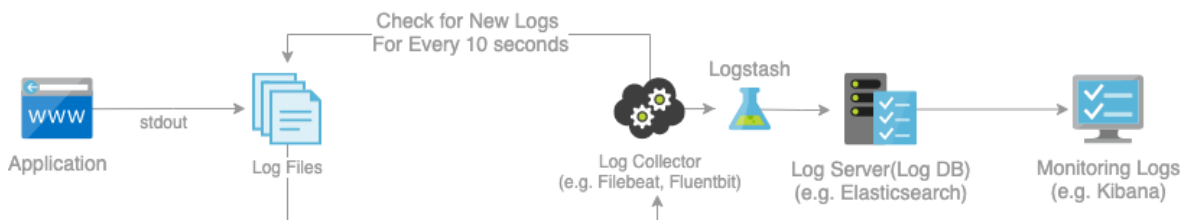


Figure 5. Logging Process In Microservices (adapted from Özgür 2019)

Microservices provide several benefits but also come with different challenges compared that require new skills and knowledge to resolve. However, as technology has been for

years, tools and solutions are mostly invented to resolve challenges for organizations. However, it is essential to consider adopting microservices based on your system size. If the scope of your system is trivial, then it's unlikely you'll gain benefits that outweigh the added complexity of building and running this type of fine-grained application (Bruce & Pereira 2019). In terms of resources, for start-up or small companies, it is better to focus their limited resources on improving the current product, before thinking about migrating to microservices. Once the organization grows bigger, it can gradually shift into microservices to gain fully benefits from it.

3 Technical Implementation Theory

In this section, we discuss the related technology to develop a microservices architecture such as containerization and container orchestration. We will elaborate on their concepts and roles in microservices, preparing for implementation.

3.1 Containerization

3.1.1 Containerization

Containerization is an act of packaging software into a container. A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. (Docker 2020.).

Packaging applications in containers benefits both development and operation side. Developers can build software locally, knowing that it will run identically regardless of host environment—be it a rack in the IT department, a user's laptop, or a cluster in the cloud. Operations engineers can concentrate on networking, resources, and uptime—and spend less time configuring environments and battling system dependencies. (Mouat 2015.)

In the implementation, thesis writer uses Docker as container runtime. However, the procedure can also apply to other runtimes such as CoreOS rkt, LXC Linux Containers.

3.1.2 Containerization and Microservices

The common procedure for containerization as followed:

- Build an image for a service
 - Run multiple instances — or containers — of your image
 - Push your image to a shared repository, or registry
- (Bruce & Pereira 2019.)

Docker can build images automatically by reading the instructions from a *Dockerfile* (Docker 2020). As images can inherit from other images, developers can choose either to inherit from public, canonical images for different technology stacks, or you can build your own base images to encapsulate standards and tools you use across multiple services. (Bruce & Pereira 2019). Secondly, developers can run images for manual and integration testing. Docker provides CLI tools to run images in local environment (Docker 2020). It also comes with parameters to define namespace and port that containers live on. However, as database containers and server containers are separated components, it is necessary to establish Docker network between them. Lastly, when containers run successfully and being tested, developers can deploy it to push a container image to an image registry. In the implementation, thesis writer uses Google Container Registry to

store image. After that, images are ready to apply to orchestrator and run on microservices!

It is worthwhile to identify some general best practices for building and naming images. Burns, Villalba, Strebel & Evenson (2019) identifies a common risk in image build process as “supply-chain attacks”. In such attacks, a malicious user injects code or binaries into some dependency from a trusted source that is then built into your application. Because of the risk of such attacks, it is critical that when you build your images you base them on only well-known and trusted image providers.

On the other hand, tag naming is also important for deployment. As service gets pushed to image register after built, it is important to keep track of images version by its tag. However, Burns, Villalba et al. (2019) suggests version tag to be immutable. For example, v1.0.1- da39a3ee can be used and if developers want to update on top of that image without changing version, v1.0.1- 5e6b4b0d is also a good example. Otherwise, “latest” tag is used as a default if there is no image version, which is convenient in development but can be challenging to manage and perform any rollback in production as “latest” tag is being mutated every time a new image is built.

3.2 Container Orchestration

Although containerization is a good way to bundle and run applications, as the system grows, it is hard to manage all containers manually. For that reason, container orchestration comes into place to provide the ability to automate these tasks. Containerization (or container scheduler) is a software tool that provides a higher level deployment platform for containers by orchestrating and managing the execution of different workloads across a pool of underlying infrastructure resources (Bruce & Pereira 2019).

There are several containers orchestrators in the market such as Docker Swarm, Kubernetes, and Apache Mesos. Kubernetes, backed by Google, has the widest mindshare and has garnered significant implementation support from other organizations, such as Microsoft, and the open-source community (Bruce & Pereira 2019). Therefore, thesis writer decides to focus on Kubernetes as container orchestrator in the thesis.

3.2.1 Kubernetes

Kubernetes is an open-source orchestrator for deploying containerized applications. The system was open-sourced by Google, inspired by a decade of experience deploying scalable, reliable systems in containers via application-oriented APIs, and developed over the last four years by a vibrant community of open source contributors. (Burns & Tracey 2018.)

Kubernetes has Pods, ReplicaSets, Services as its core objects. A pod is the unit of deployment and represents a single instance of a service (Burns & Tracey 2018). Because it's the unit of deployment, it's also the unit of horizontal scalability (or replication). When you scale capacity up or down, you add or remove pods. (Bruce & Pereira 2019). On the other hand, Pods keep application resilient by automatically restart in case of any container's crashes, which creates first level defender for application in case of failure. This practice is also managed by Kubernetes automatically. However, there is a possibility to have multiples containers within Pod if they are tightly connected. In fact, Pods are designed to have containers inside it connect on localhost, which is "ideally suited for symbiotic relationships between their containers, such as the main serving container and a background data-loading container". (Burns & Tracey 2018)

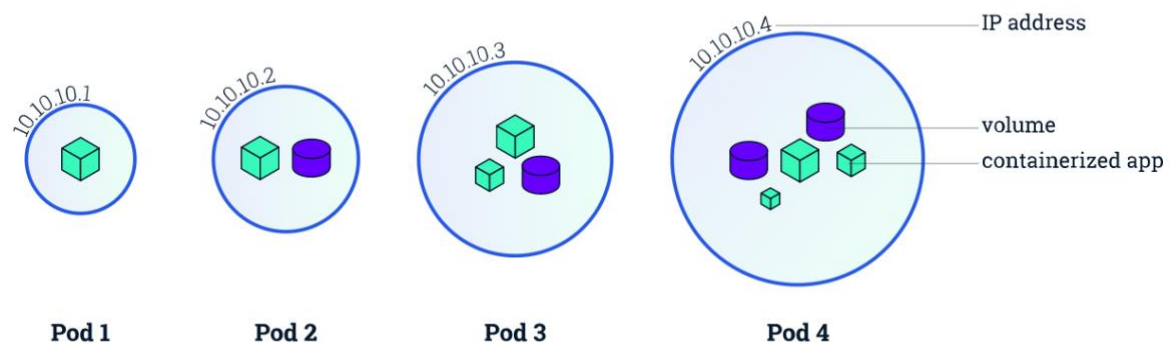


Figure 6. Pod and Container (adapted from Kubernetes.io 2020)

For replicating containers into multiple instances, Kubernetes provides ReplicaSets object. Burns & Tracey (2018) depicts the benefits of replication as although individual containers may fail or may be incapable of serving the load of a system, replicating an application out to several different running containers dramatically reduces the probability that your service will completely fail at a particular moment in time. In fact, by defining number of replicas in YAML file, Kubernetes controller manager ensures that replication is performed and a specified number of Pod replicas are running at any given time.

After replicating service using a replica set, Kubernetes provides Service object to handle networking. The Service load balancing is programmed into the network fabric of the Kubernetes cluster so that any container that tries to talk to the Service IP address is correctly load balanced to the corresponding Pods (Burns & Tracey 2018).

When a service is created, it receives its own internal IP address, DNS entry and load-balancing rules that proxy traffic to the Pods that implement the traffic. This fixed IP represents all replicas of the service. This IP address is virtual—it does not correspond to

any interface present on the network. Instead, it is programmed into the network fabric as a load-balanced IP address. When packets are sent to that IP, they are load balanced out to a set of Pods that implements the Service. (Burns & Tracey 2018.) In case of failure or scaling of a Pod, load balancer constantly reprogrammed to match the current state of the cluster. Therefore, clients can rely on the connections to Service IP address provided to a Pod implementing the Service.

3.2.2 Kubernetes and Microservices

To successfully deploy container in Kubernetes, it is important to understand orchestrator's workflow.

At a high level, a container scheduler workflow looks something like this:

- Developers write declarative instructions to specify which applications they want to run. These workloads might vary: you might want to run a stateless, long-running service; a one-off job; or a stateful application, like a database.
- Those instructions go to a master node.
- The master node executes those instructions, distributing the workloads to a cluster of underlying worker nodes.
- Worker nodes pull containers from an appropriate registry and run those applications as specified.

(Bruce & Pereira 2019, Chapter 9.)

As such, one defines a set of desired states in *deployment manifest* and Kubernetes will handle the rest. To simplify the process, defining deployment manifest consists of two parts: Deployment and Service. In Deployment, developers first declare a number of replicas template for Pod. This is to make sure that the number of Pod replicated is consistent and they are all similar. Then, developers define container image that the Pod uses and specifies name, port and environment variables for that Pod. Note that it is also a good practice to store important credentials here to avoid any breach. Figure 9 depicts a common Deployment.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      containers:
        - name: app
          image: thitrandocker/hello-kubernetes:1.0.0
          ports:
            - containerPort: 3000
          env:
            - name: MONGO_URL
              value: mongodb://mongo:27017/dev
          imagePullPolicy: Always

```

Figure 7. Defining Deployment Kubernetes (adapted from Polencic 2019)

However, Deployment for database service has to concern about database consistency. Pods are designed to be able to restart when needed, but database should stay consistent. In fact, storage must persist in any circumstances. Therefore, database service has to place in persistent storage volume and Kubernetes provides PersistentVolume and PersistentVolumeClaim for this reason. (Polencic 2019.) As such, Deployment of database consists of three resources: PersistentVolumeClaim, Service and Deployment.

As Deployment defines how to run an app, it needs Service to make it available for usage. There are three common types of the Service: ClusterIP, NodePort and LoadBalancer. Figure 10 depicts these types and their behavior.

Service Type	Behavior
ClusterIP	Exposes the service on an IP address local to the cluster
NodePort	Exposes the service on a static port accessible at the cluster's IP address

LoadBalancer	Exposes the service by provisioning an external cloud service load balancer (If you're using AWS, this creates an ELB.)
--------------	---

Figure 8. Types of Service on Kubernetes (adapted from Bruce & Pereira 2019)

Because database should not be exposed to outside of the cluster, it should have ClusterIP as Service Type. In fact, ClusterIP is also the default type of Service so developers do not need to specify this. For Application, developers can decide to have NodePort or LoadBalancer depending on the environment. On top of that, developers define name and port where the service exposes to. Figure 11 illustrates common Service manifest.

```
apiVersion: v1
kind: Service
metadata:
  name: hello
spec:
  selector:
    app: hello
  ports:
    - port: 80
      targetPort: 3000
  type: LoadBalancer
```

Figure 9. Defining Service Kubernetes (adapted from Polencic 2019)

(Note that *spec.selector.app* is similar to label from Deployment)

In GKE, there is also a new concept of cluster. A cluster consists of a pool of Compute Engine VM instances running Kubernetes, the open-source cluster orchestration system that powers GKE. (Google Cloud 2020).

Figure 12 depicts the overall architecture of Microservices using Kubernetes and Docker.

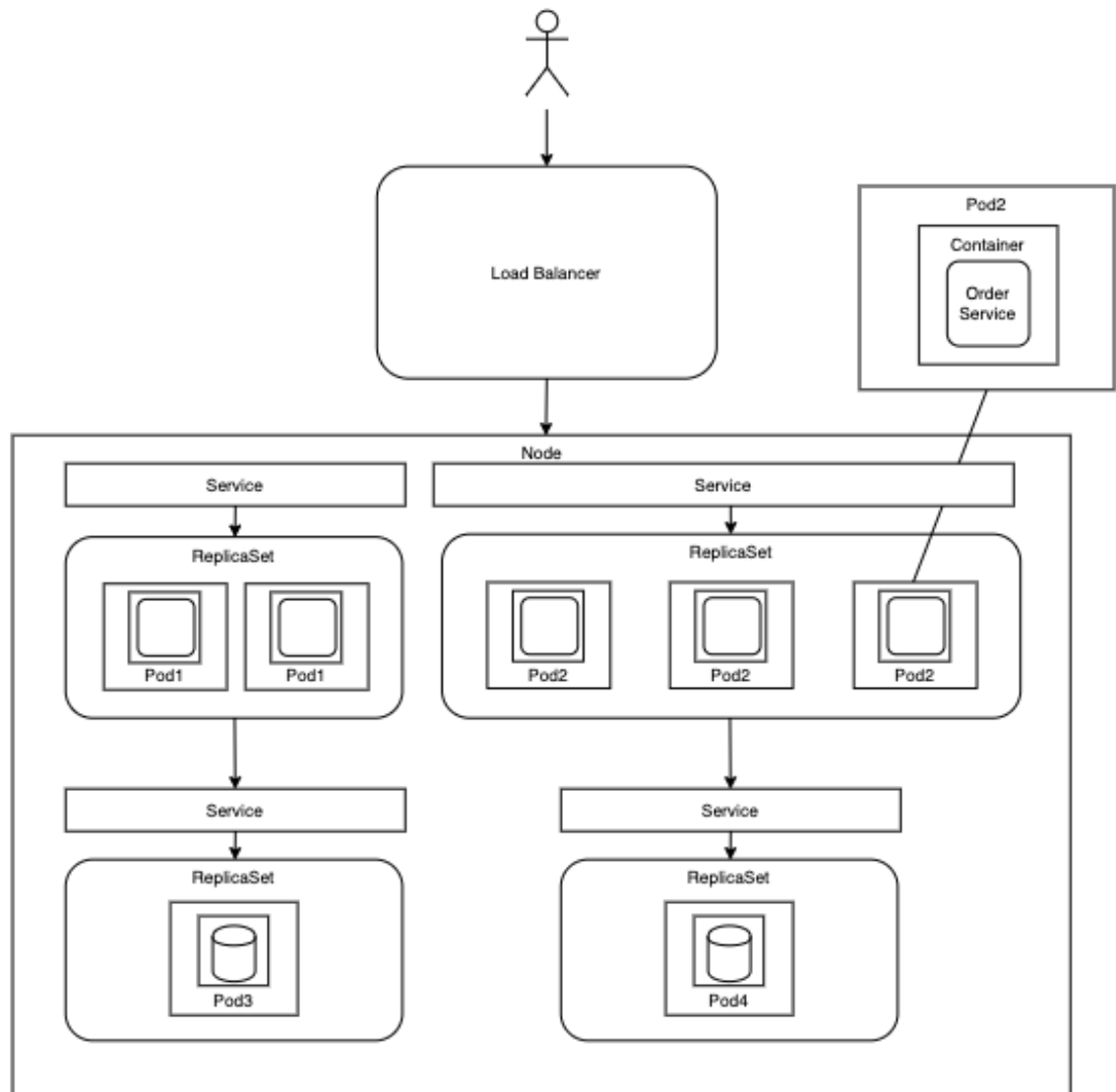


Figure 10. Microservice with Kubernetes and Docker (adapted from Kubernetes.io s.a.)

3.2.3 Services Communication in Kubernetes

It is also essential to understand how connections are handled in Kubernetes with Docker containers. For containers within the same pod, because “each container in a Pod can see the other containers in the Pod on *localhost*” (Burns & Tracey 2018) and “Docker uses host-private networking, so containers can talk to other containers only if they are on the same machine” (Kubernetes.io 2020b), these containers can simply communicate with each other via port on *localhost*.

How about communicating between containers that are in different Pods? It is tempting to just talk to the pods directly. However, this is a bad practice as Kubernetes.io (2020b) states that when a node dies, the pod dies with it, and Deployment will create new ones, with different IPs. As such, it is better to use Service for communication between

containers in different Pods. In fact, when created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. (Kubernetes.io 2020b.) And as Pods talks to a Service, it will also load-balance out to some member Pods of that Service, and thanks to Service defining its IP address and port, containers can curl to it on `<IP-address>:<port>` from any node in the cluster.

Sometimes services need to communicate with each other to fulfill a job. For example, if an order is placed, not only Order service needs to handle the order, but also Fees service needs to charge fee, Account Transactions service needs to reserve stocks for reservation and User service might need to record order information. As the application gets more complicated, communication between services become harder to manage. More importantly, services in microservices should be independent deployment units, where one service fails will not affect other services. Therefore, one challenge in communication is to balance between having communication between services while keeping these services loosely coupled. The following chapter will introduce service domain and shared model method to keep services highly cohesive and loosely coupled.

4 Implementation Framework

In this section, we discuss framework of migration process to microservices. We will also provide a walkthrough to create a service as well as to containerize and orchestrate it in microservices environment.

4.1 Service Domain

Services in microservices need to well model around its domain. In fact, services should be highly cohesive and loosely coupled. Newman (2018, Chapter 3) defines loosely coupled as a change to one service that should not require a change to another. This is the de factor of independent deployability of services in microservices as discussed above in Chapter 2. This also limits as much as possible different types of calls between services, which can lead to a potential performance problem. On the other hand, Newman (2018, Chapter 3) also defines high cohesive as services with related behavior to sit together, and unrelated behavior to sit elsewhere. This can also be considered as a module in monolithic, where code with similar behavior lives in one place. Generally, highly cohesive and loosely coupled services mean having codebase with related behavior in one service and communication between service being as loosely as possible. The visualization can be shown in figure 13.

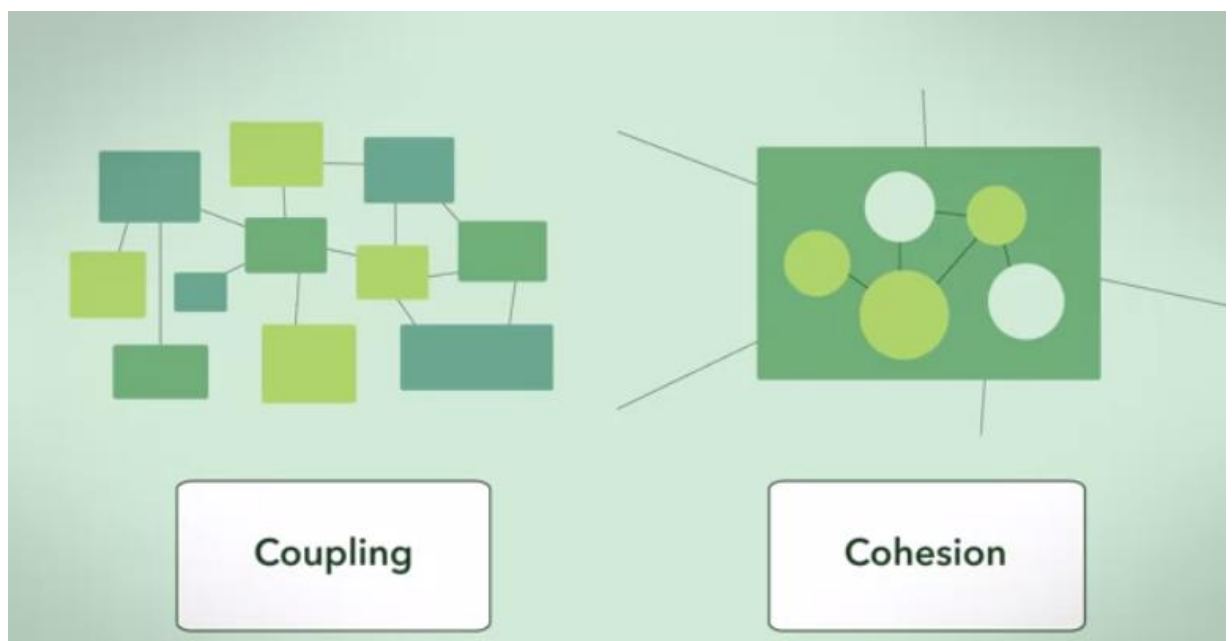


Figure 11. Loosely coupled and highly cohesive services (adapted from Wong 2019)

To achieve this, having service boundaries is the first priority and Bounded context is a good system for it. What is bounded context? Evans (2003) depicts as it delimits the applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it relates to other Contexts. In microservices, any given domain consists of multiple bounded contexts, and residing

within each are things that do not need to be communicated outside as well as things that are shared externally with other bounded contexts. Each bounded context has an explicit interface, where it decides what models to share with other contexts. (Newman 2018.)

For example, with Order and User service, User service within User Domain has multiple bounded contexts within it such as Info, Clicking Behaviors, Password Recovery, etc. as well as History, Report, Billing in Order service. They both should have an explicit interface to the outside world (Report, Info, etc.) and models that only they need to know about (Clicking Behaviors, History, Password Recovery). Order service only needs to know about User Info for writing bills, but neither Recommendation nor Password Recovery. Likewise, Billing in Order should not be shared externally either.

Hence, User Info is called a shared model between the two contexts. However, not everything from User Info should be shared. For example, User Info such as password and credits card info does not need to be exposed in the shared model.

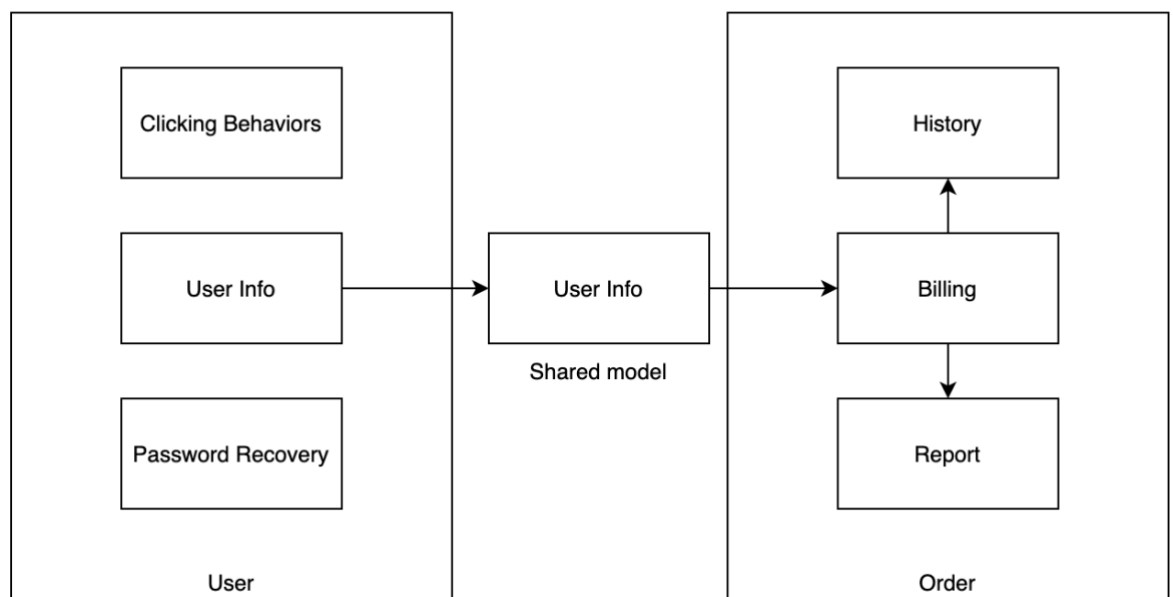


Figure 12. Shared model between User and Order Service (adapted from Evans 2003)

Bounded context helps to build services that are highly cohesive and loosely coupled. By defining what models should be shared, and not sharing internal representation, services avoid resulting in tight coupling. In fact, services now have only information it needed from a shared model which avoids dependent between services and potential data breach. On the other hand, each service, within its bounded context, has all models which serve business objectives that domain should cover, leading to high cohesiveness in services.

It is important to notice that as new service now becomes an individual binary, it requires to have its own framework, dependency management system, architecture and file organization. In fact, one should consider design pattern to organize the code and importantly, as service grows, it might need to split itself into other services. Therefore, it is good practice to keep the code clean and think about writing code in "Strangler way" so that it will be easier on the other end. Thesis writer suggests looking at MVC design pattern (or MVCS specifically) – a standard design pattern for Java application - as it is effective and relatively easy to follow. Moreover, new services should be treated as an individual application. The notion of whether or not to have new repositories for new services at this stage is controversial, but new services should have their pipelines for continuous integration. This is crucial to make sure that new services are working as expected together with the existing system. However, the implementation of CI pipeline falls out of scope in the thesis.

4.2 Strangler Pattern

Migrating into microservices can be overwhelming in the first place. Suddenly, not only all codebase has to be changed and divided into smaller services, but also database has to migrate to live under each service. To resolve it, several methods have been invented. Strangler Application Pattern (will be referred to as Strangler pattern later on) is one solution for migration process, which was also introduced by Google, IBM in their own development process. (Brown 2017; Felix 2019)

Strangler Process was invented by Martin Fowler by watching nature occurrence. He got inspired by watching strangler figs in rain forest. The seed in the upper branches of a tree and gradually work their way down the tree until they root in the soil. Over many years they grow into fantastic and beautiful shapes, meanwhile strangling and killing the tree that was their host. (Fowler 2014.) In fact, Strangler is mostly used in rewriting systems. In that context, it is to gradually create a new system around the edges of the old, letting it grow slowly over several years until the old system is strangled. (Fowler 2014)

Implementation of Strangler pattern is applicable for microservices migration with slight modification. Firstly, monolithic system divides into domains representing each service, where bounded context system from the previous chapter can be a good method for the process. Secondly, monolithic systems can either reuse its existing load balancer or create a proxy for redirecting traffics. This step is very important as it creates a transition bridge for services in monolithic to transfer to microservices system. Then, each by each, services after divided into monolithic transfers into a new microservices system with its own database. With each service transferred, proxy redirects traffic to that service of monolithic system into that service in the new system. Gradually, all services from

monolithic system are transferred into a new system. When finishing extracting services to microservices, the process is done with a new microservices system and developers can completely jump to the new system and enjoy the benefits of microservices. (Walls 2019.) Figure 15 illustrates migration process using Strangler pattern.

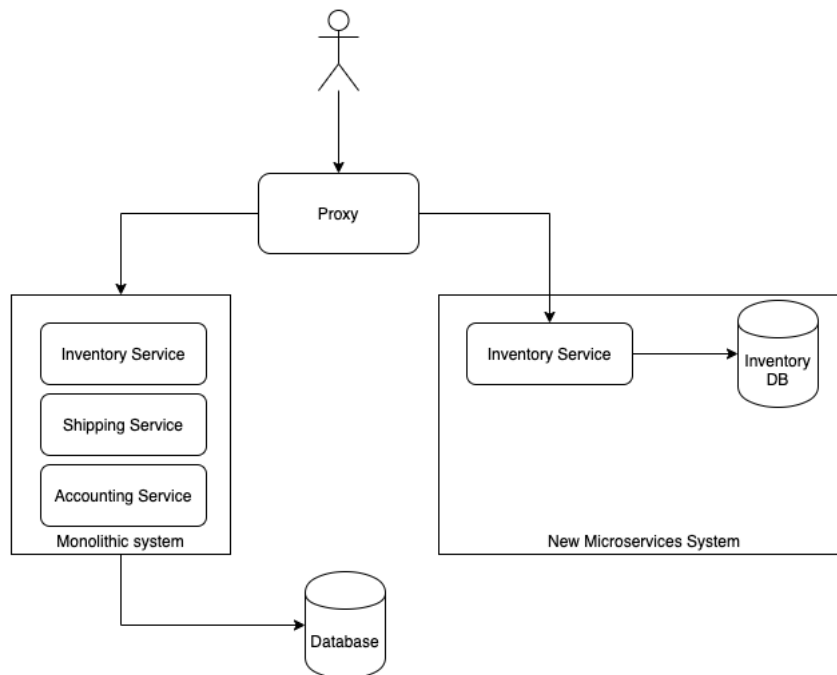


Figure 13. Implementing Strangler Pattern to Microservices Migration (adapted from Walls 2019)

Strangler Pattern creates incremental development for migration process instead of just one big update. Brown (2017) depicts its value as it creates incremental value in a much faster timeframe than if you tried a “big bang” migration in which you update all the code of your application before you release any of the new functionality. In fact, if implementing properly, the system as a whole can still work without users noticing changes. It also gives you an incremental approach for adopting microservices—one where, if you find that the approach doesn’t work in your environment, you have a simple way to change direction if needed. (Brown 2017) On the other hand, Strangler Pattern opens the possibility to have migration process go hand in hand with developing a new feature. As everything new service is transferred into new microservices, teams have enough time for adding a new feature or refactor code if needed.

Brown (2017) depicts an anti-pattern while applying this method as *Don’t apply it one page at a time*. As migration is a challenging process, it is tempting to just go halfway as to only transfer the codebase without replicating its own database. This practice is bad as it allows two different data access at the same time, which creates data inconsistency in the future. Moreover, for adding new features while transferring services, it is important to

create a new feature with tests case to make sure it does not break the system. Moreover, it is also not a good practice to add many features into existing service in migration process as it can prolong the timeline and lessen motivation for migration process.

5 Implementation - Migration to Microservices

In this section, we will implement Strangler Pattern in a process of transforming an eCommerce store from monolithic to microservices. We will introduce a monitoring system for running during the migration. Based on data collected from the monitoring system, we can measure availability of the system and draw a conclusion on how using Strangler Pattern can help to keep migration process stable and reducing risks.

5.1 Project background

In this implementation, an eCommerce store application will be used as a target of migration process, specifically its backend codebase. The application can be considered as a *tori.fi* for bakery. Any user can register to become a Baker. They can use their credentials (email, password) to sign in. Then, they will have their own Baker page, where they can post cakes that they can make. On the other hand, users, like Customers, can search for the type of cake that they want, defining type, city and pickup date. When the Customers search, the application returns a list of bakers that has products related to the type of cake, having the same city and pickup date within their availability day. Customers can now select from any baker from the list and place an order for them. When Customers complete their order, the application will send an email to the Bakers to confirm that they will take the job, then notice Customers when the cake is ready. On top of that, there is an Admin who can manage all orders and Bakers. Figure 16 illustrates the jobs of each identity.

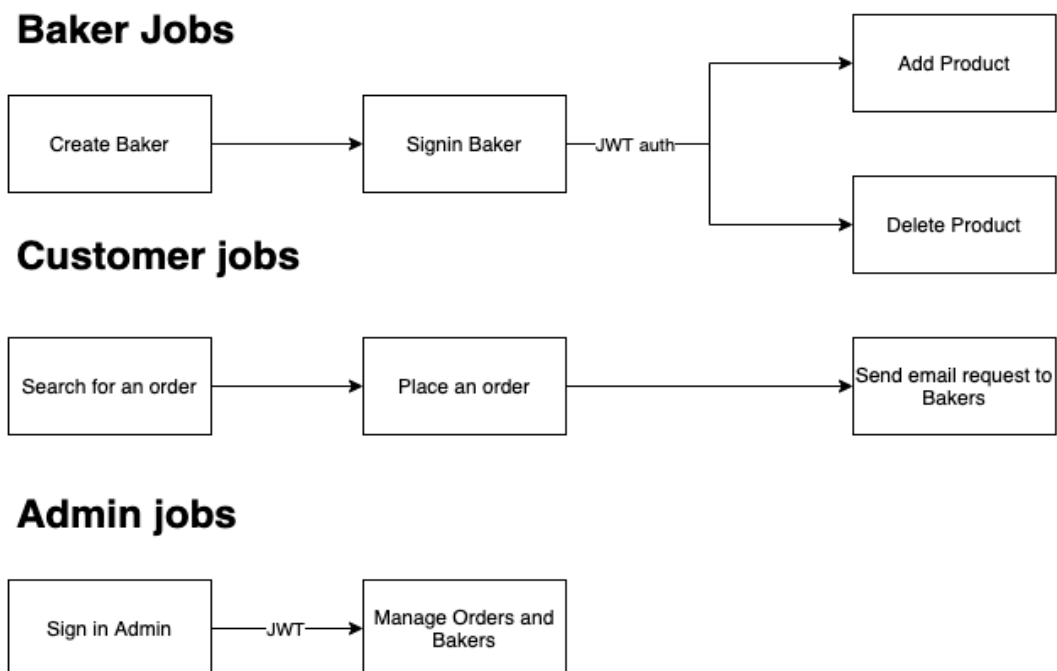


Figure 14. Jobs of Identities in Ecommerce Application

Currently, the eCommerce store is a monolithic application, which runs on Google Compute Engine. The application uses NodeJS/Express for server codebase and MongoDB as database. On top of it, it uses Process Manager package (pm2) to keep the application and database running.

5.2 Implementation plan

The implementation, the main working method will be a hands-on case study that will include conducting migration process from monolithic application to microservices of an eCommerce store. During the migration process, there will be a monitoring process sending requests to the system every 5 minutes to collect quantitative data about availability and downtime of the current system.

The following sources and tools will be used in the implementation:

- Collected quantitative data from Postman
- Splitting monolithic application to microservices using service domain
- Migration process based on Strangler pattern
- Apache Web Server for proxying traffics
- Docker for containerization
- Kubernetes for container orchestration
- Ingress as a load balancer
- Google Cloud Platform and its ecosystem

The implementation steps are based on Strangle pattern, which includes:

- Splitting monolithic application to microservices
- Set up proxy system
- Develop new services
- Handling connections

For monitoring process, Postman uses a full cycle of endpoints in the application: Create Baker, Baker Sign in, Add Product, Delete Product, Search Cake, Order, Admin sign in and Delete Baker. These requests are chained with the environment variable used to save credentials for the next request. For example, token from Baker Sign in endpoint is used to as Authorization token for Add Product endpoint. Moreover, thesis writer tests each request by its status code and return value, so that it returns the response as expected. These endpoints are grouped into a collection and being monitored by Postman over time. The point is to see how the application's health and performance over the time of the migration.

Data from Postman is analyzed to determine availability and causes of downtime during migration process. Based on the result, thesis writer draws conclusions and feedback about Strangler pattern as well as migration process.

5.3 Splitting monolithic application into microservices

At first, using Domain Driven Approach, thesis writer splits monolithic application into six services: Baker Service, Product Service, Search Service, Order Service, Auth Service, and Emailing Service. The reason is to split the application into different service domains that serve one individual objective. Figure 17 depicts the draft plan for separation.

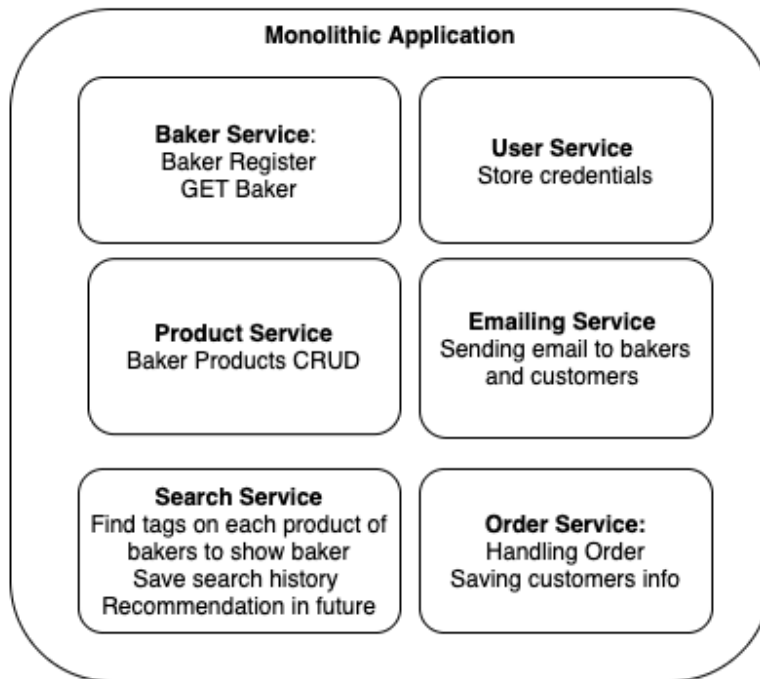


Figure 15. Splitting Monolithic Application Blueprint (draft)

However, there can be some problems with Baker Service, Product Service, and Search Service. As Search Service searches for cake based on product tag, city, and availability day, Search Service has to make requests to both Baker Service and Product Service for every request. This is a sign of *tight coupling services* as Search Service is dependent on Baker Service and Product Service.

On the other hand, searching request is synchronous communication. In every request, Searching Service request needs to find products that have a tag that is similar to the type of cake. Then, Searching Service needs to call Baker Service to return bakers from which the products belong to. If any step in the process fails, the Searching Service is responsible for initiating roll back with other services.

Are Order Service and Emailing Service also dependent? Order Service indeed has to talk to Emailing Service, but their relationship is not entirely dependent. The shared model in figure 18 depicts the relationship between them.

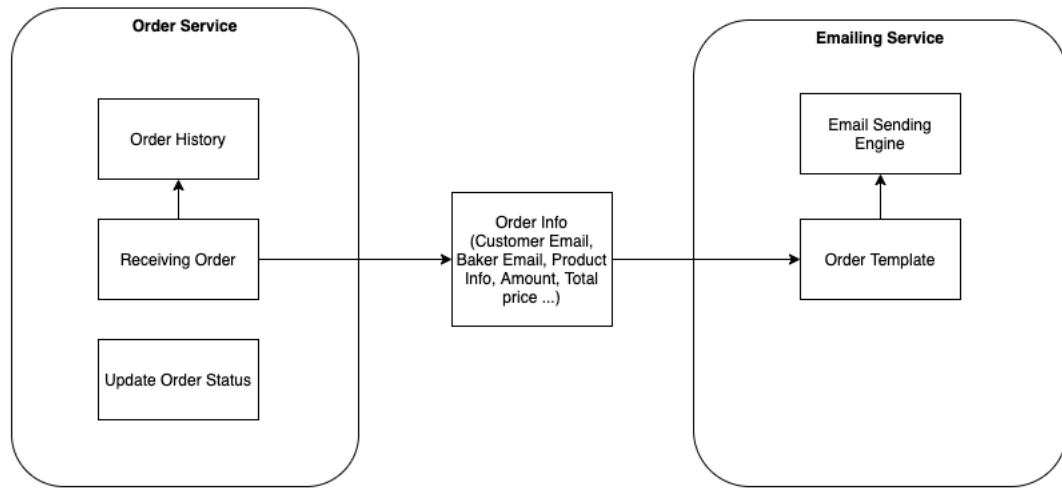


Figure 16. Order Service and Emailing Service Shared Model

As shown in Figure 18, Order Service and Emailing Service have their models but share Order Info. In fact, Order info only contains essential information about Order such as customer email, baker email, product info as Emailing Service does not need to know about other things such as credit card information from Order Service. On the other hand, as communication between services is event-driven, if Email Sending Engine model is down, which causes Emailing Service crashes, Order still can be placed as Order Service might be up. As such, by defining Shared Model with defined separated responsibility, Order Service and Emailing Service are highly cohesive and loosely coupled services.

As such, thesis writer decides to establish four services: Baker Service, Order Service, Auth Service, and Emailing Service. As such, the shared database is also split into smaller databases that can only be accessed by its service. The purpose behind this is to split the monolithic into highly cohesive and loosely coupled services. The tradeoff is that Baker Service can be “big” at the moment. However, it can be a good project to split Baker Service into smaller services later, keeping in mind during the migration process to design Baker Service for this purpose. Figure 19 and 20 depict the blueprint for migration process.

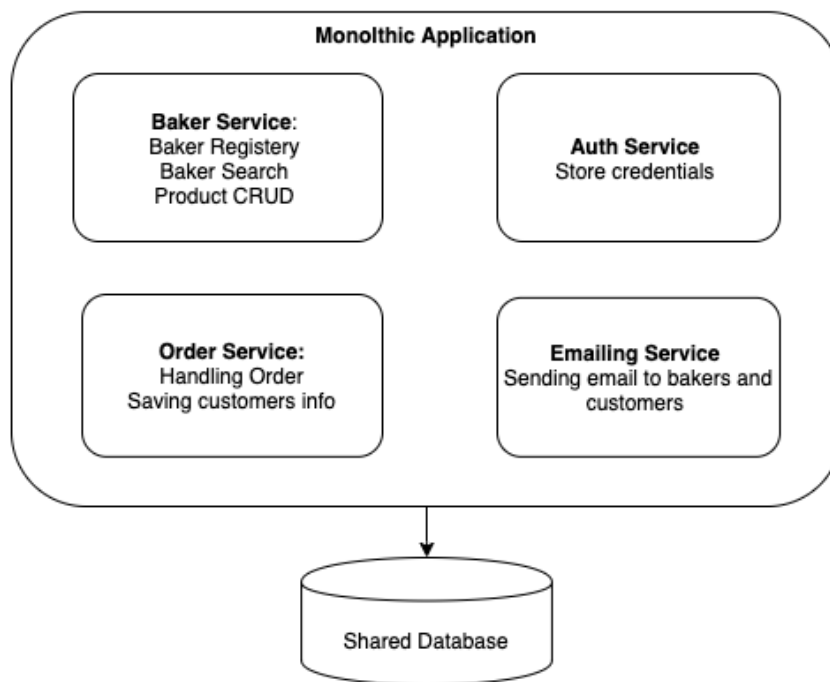


Figure 17. Splitting Monolithic Application Blueprint (approved)

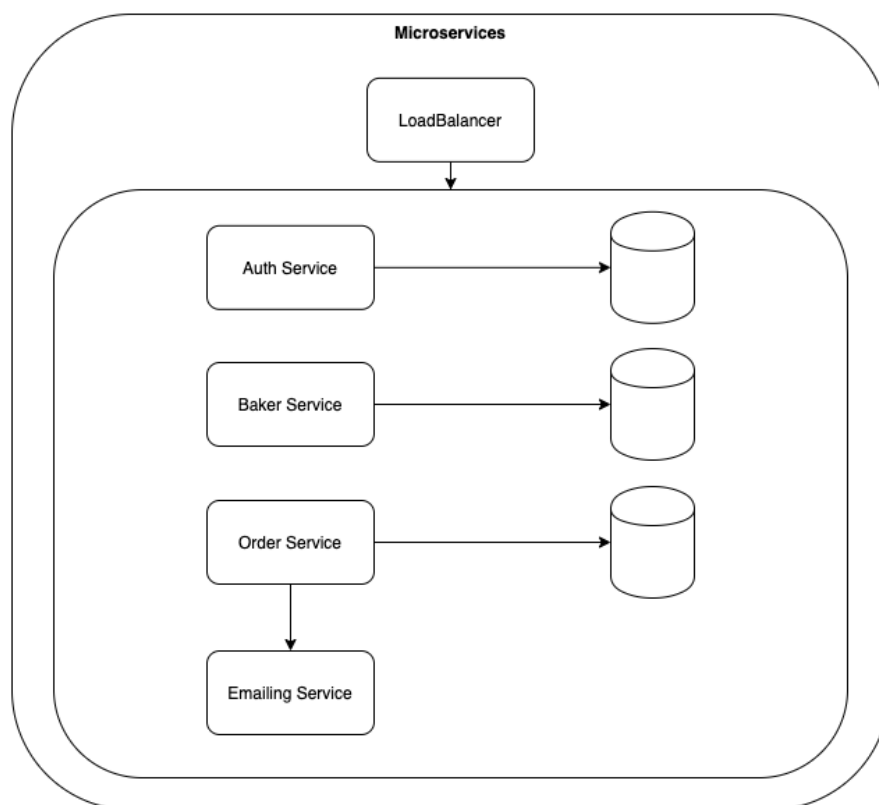


Figure 18. Microservices Implementation

As such, with the blueprint finished, the next step is to implement Strangler Pattern to migrate monolithic application to microservices. Also, it is a good time to start monitoring the system!

5.4 Setting up a proxy system

The second step for the implementation of Strangler Pattern is to create a proxy system to redirect traffic. This proxy acts as a transitional bridge to redirect any traffics from monolithic application to new services created during the migration process.

This objective suits the purpose of a *reverse proxy*. Papiernik (2017) defines reverse proxy as a type of proxy server that takes HTTP(S) requests and transparently distributes them to one or more backend servers. Following his blog for Digital Ocean, thesis writer sets up “Apache as a basic reverse proxy using the mod_proxy extension to redirect incoming connections to one or several backend servers running on the same network. “ (Papiernik 2017). However, at this point, reverse proxy only redirects all traffic to monolithic application as new services have not been developed.

The application needs to set up Apache Web Server if it is not installed by default. From SSH terminal of EC2 instance:

```
sudo apt-get update
sudo apt-get install apache2
```

Then, enable mod_proxy and proxy_http module.

```
sudo a2enmod proxy
sudo a2enmod proxy_http
```

To put these changes into effect, restart Apache:

```
sudo systemctl restart apache2
```

Apache is now ready to act as a reverse proxy for HTTP requests. Next, set up Apache virtual host to serve as a reverse proxy for monolithic application.

```
sudo vim /etc/apache2/sites-available/000-default.conf
```

Replace all contents within Virtual Host block with the following:

```
<VirtualHost *:80>
    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:8080/
    ProxyPassReverse / http://127.0.0.1:8080/
</VirtualHost>
```

http://127.0.0.1:8080 is the address where monolithic application is running on locally. If there are other addresses for the application, use their addresses instead.

- ProxyPreserveHost makes Apache pass the original Host header to the backend server. This is useful, as it makes the backend server aware of the address used to access the application.
- ProxyPass is the main proxy configuration directive. In this case, it specifies that everything under the root URL (/) should be mapped to the backend server at the given address. For example, if Apache gets a request for /example, it will connect to `http://your_backend_server/example` and return the response to the original client.
- ProxyPassReverse should have the same configuration as ProxyPass. It tells Apache to modify the response headers from the backend server. This makes sure that if the backend server returns a location redirect header, the client's browser will be redirected to the proxy address and not the backend server address, which would not work as intended.

(Papiernik 2017.)

Hence, if there is a change to URL pointing to a specific address in ProxyPass and ProxyPassReverse, it would result in traffics to that URL redirected to that specific address. For example, the following directives will redirect traffic for URL `/order` to `127.0.0.1:3000`.

```
<VirtualHost *:80>
    ProxyPreserveHost On
    ProxyPass /order http://127.0.0.1:3000/
    ProxyPreserveHost /order http://127.0.0.1:3000/
    ProxyPass / http://127.0.0.1:8080/
    ProxyPassReverse / http://127.0.0.1:8080/
</VirtualHost>
```

However, the application does not need to redirect traffic yet so it should remain as the previous state.

Then, restart Apache to put these changes into effect.

```
sudo systemctl restart apache2
```

As such, a proxy system is ready to redirect traffics to newly created services in the next section.

Now, as the base work is ready, the next step is to create new services based on migration blueprint. It is reasonable to tackle services from the simplest to the most complicated to avoid too many changes and unexpected behavior from the system. Thesis writer proposes the procedure: Emailing Service, Order Service and Baker Service. Thesis writer writes about the migration of Emailing Service as illustration.

5.5 Developing new services

In monolithic application, all files of the service in one module called *emailings*, which makes it easier to build it as a separate service. This is a good example to keep in mind when building new services in microservices. Codebase should be

written in a fashion that it can easily decouple later on. To create a new separate service, copy all the files from *emailings* module and its metadata to a new directory.

Before deploying new Emailing Service to microservices, one needs to containerize it. The following steps are to define and push Emailing container to Google Container Register.

```
docker build -t emailing-service .
```

First, set PROJECT_ID environment to Google Cloud project ID (project-id)

```
export PROJECT_ID=project-id
```

Configure Docker to authenticate to Container Registry (only need to run once)

```
gcloud auth configure-docker
```

Push image to Google Container Registry.

```
docker push gcr.io/${PROJECT_ID}/emailing-service:v1.0
```

The script above deploys emailing application to cluster using its container image. It also exposes application with Kubernetes Service to external traffic with port 80. At this stage, the service type is LoadBalancer because it still requires to receive inbound traffic to perform the job. As other services emerge in microservices application, service type change can change to ClusterIP later on.

First, to run container image, create a cluster in Google Cloud Shell.

Set your project ID and Compute Engine zone.

```
gcloud config set project $PROJECT_ID  
gcloud config set compute/zone compute-zone
```

Create a two-node cluster named ecommerce-microservices to separate from ecommerce-monolithic.

```
gcloud container clusters create ecommerce-microservices --num-nodes=2
```

Apply Kubernetes manifest file stored in kube directory

```
kubectl apply -f kube
```

To see Pods created by Deployment

```
kubectl get pods
```

5.6 Handling Connections

The next step is to set up Kubernetes Service to start receiving traffics to Emailing app. This can be done in manifest or using a command shortcut from kubectl.

```
kubectl expose deployment emailing-app-service --type=NodePort --  
port 80 -target-port 8080
```

The service type is NodePort because Emailing app should not expose itself directly outside. Instead, we want that external traffics will have to go through a Load balancer, then it will redirect to each service based on specific endpoints. We use Ingress for this purpose. Configure a manifest at /infra/loadbalancer.yaml.

```
apiVersion: networking.k8s.io/v1beta1  
kind: Ingress  
metadata:  
  name: loadbalancer  
spec:  
  rules:  
  - http:  
    paths:  
    - path: /sendEmail  
      backend:  
        serviceName: emailing-app-service  
        servicePort: 80
```

Google Cloud (2020) states that a Service exposed through an Ingress must respond to health checks from the load balancer. Therefore, we have to add an endpoint for health check /healthy which has a response with status 200. On top of it, configure Kubernetes Deployment manifest file.

```
...  
readinessProbe:  
  httpGet:  
    path: /healthy  
    port: 3000
```

With port is the port that container runs on inside Pod. For Emailing app, it is 3000. Apply new configuration

```
kubectl apply -f Emailing/kube
kubectl apply -f infra
```

Check the final result.

```
kubectl describe ing
```

The result should be as

```
Name:          loadbalancer
Namespace:     default
Address:       34.107.142.32
Default backend: default-http-backend:80 (10.12.0.7:8080)
Rules:
  Host          Path  Backends
  ----          -
  *
                /sendEmail  emailing-app-service:80 (10.12.0.14:3000)
Annotations:  ingress.kubernetes.io/backends: {"k8s-be-32048--
c650713c443f0c00":"HEALTHY","k8s-be-32150--c650713c443f0c00"
:"HEALTHY"}
              ingress.kubernetes.io/forwarding-rule: k8s-fw-default-
loadbalancer--c650713c443f0c00
              ingress.kubernetes.io/target-proxy: k8s-tp-default-
loadbalancer--c650713c443f0c00
              ingress.kubernetes.io/url-map: k8s-um-default-loadbalancer-
-c650713c443f0c00
Events:
  Type          Reason    Age   From                      Message
  ----          -
  Normal        ADD       57m   loadbalancer-controller   default/loadbalancer
  Normal        CREATE    56m   loadbalancer-controller   ip: 34.107.142.32
```

As such, emailing application is ready to serve with IP address 34.107.142.32 with endpoint /sendEmail. Now, monolithic application can use it to send requests for sending email. Once everything is running, emailing module from monolithic application can be deleted.

Then, monolithic application redirects traffics for emailing application to the new service. (at this stage, when an order is placed, it will send requests to email service for sending emails)

```
<VirtualHost *:80>
  ProxyPreserveHost On
  ProxyPass /sendEmail http://34.107.142.32/sendEmail
  ProxyPreserveHost /sendEmail http://34.107.142.32/sendEmail
  ProxyPass / http://127.0.0.1:8080/
  ProxyPassReverse / http://127.0.0.1:8080/
</VirtualHost>
```

Then, restart Apache server to apply the new configuration.

```
sudo systemctl restart apache2
```


Following this example, each service from monolithic application can follow the circle Setup Proxy, Developing New Service, Handling Connection for each by each to migrate to microservices.

5.7 Conclusions

5.7.1 Reflections on monitoring data

Overall, the migration process occurred in one week, starting from setting GKE environment, to monitoring and finding the right tools.

The main objective is to measure the availability time of the system during migration process. Postman, the main tool that is used for monitoring process, has proven to perform quite well throughout the migration process. However, due to the limitation of software, it can only send requests every 5 minutes. The work assumes that service downtime is roughly 5 minutes when a request fails. In a better world, requests from Postman can also be integrated with other tools to measure the exact downtime to seconds. However, with the task of measuring availability of system over the migration period, Postman has provided descriptive picture of how the system performs through metrics such as the number of success/fail responses, success/fail tests as well as response time and failed percent over time. Based on the metrics, data has been collected as in figure 21.

Metrics	Data
Downtime	30m
Total period	19h30m
Availability	97,4%
Requests sent	640
Failed request	19
Success rate	97%

Figure 19. Monitoring data from Postman

However, it is essential to notice that most of downtime is due to user error. As Postman sends a request every 5 minutes, they are 6 failed attempts in total 30 minutes downtime. The reasons behind them being:

- In the beginning, there were 2 fail efforts due to user error. As the first time setting up the environment, variables were mistaken, which led to incorrect requests.
- In the third failed attempt, there was already Baker with existing email in the database due to previous requests while building new service, which causes the

request to create baker and search cake were failed. This is proven because baker login, add product, delete product, etc. were all working.

- More importantly, the fourth failed attempt was due to restarting proxy from monolithic application after redirect for Auth service traffic to new service in microservices. As the proxy system was reconfiguring, all requests were unsuccessful.
- The fifth and sixth were due to library mismatch. JWT token generated in new Auth service cannot be used in monolithic application.

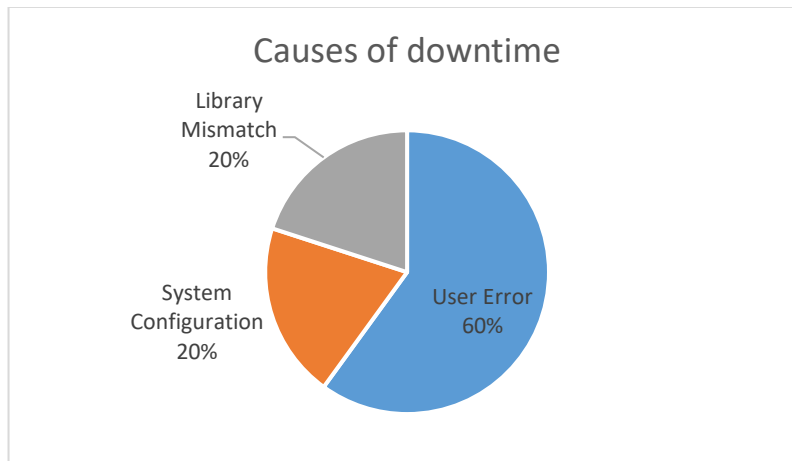


Figure 20. Causes of system downtime

In conclusion, based on data collected, implementation of Strangler pattern in the migration process from monolithic application to microservices has shown availability of 97,4% and successful requests rate is 97%. In 2,6% downtime of the system, 80% of that is caused by user error and library mismatch, whereas only 20% is from system configuration.

5.7.2 Reflections on migration framework

Strangler pattern, the main framework for transforming from monolithic application to microservices, has proven its benefits in migration process. As each service is decoupled from monolithic application to microservices (starting from the easiest one), microservices has evolved with an incremental added value over time. This helps to avoid the complexity involved in the relations of multiple services and keep things simple at the beginning. Instead of building a complex system of multiple services, thesis writer started with the easiest service – Emailing service – and then moved on to Order service, Auth service, and finally Baker service. This procedure is useful because the first service is not necessarily complicated in itself, but instead challenging to set up new microservices environment for it. In fact, thesis writer spent plenty of time to find the right tools for setting up Kubernetes environment, finding the right component such as Load Balancer (thesis writer found and tested multiple tools such as API gateway, Google Cloud Functions,

Service Mesh Istio but ultimately decide Ingress as it is the simplest and most effective option). This can relate to organizations when they first migrate into microservices. Therefore, Strangler Pattern helps simplify migration process at the beginning, which prepares rooms for more important tasks such as setting up environment. As migration process continues, new services are more complicated to extract from monolithic application, but organizations have more time for it as the environment is already familiar for them. Thesis writer has the same experience with simple service and challenging environment setup at the beginning, whereas challenging service's migration and familiar setup later on.

Strangler pattern also shows its effectiveness in keeping the migration process resilient and avoiding risks. With over 97% availability during migration process (taken into account user error and library mismatch configuration) and 97% success request, Strangler Pattern provides a good foundation for organizations to consider migrating into microservices with safety.

While Strangler pattern has proven to work, it is obvious that it is a one-rule-all choice for migration process. Luckily, Strangler pattern works very well with small and medium-sized systems which does not involve much complexity; however, for a more complicated system, Strangler pattern has shown some of its concerns. Firstly, the lack of will or resources to finish the strangling process can be the main reason failed project. The more complicated the system, the more time needed to extract services to microservices. Hence, the migration process requires more resources and will to finish the project. Otherwise, one monolithic system can result in two complicated, awkward system, which might be worse than before it started. Moreover, for large organizations, communication is key in migration process. It is important to keep consensus between team members (if possible everyday communication) to keep everyone on the same page of Strangler process, avoiding misunderstandings or disagreements in the future of what is service boundary and what is not.

As migration to microservices is a complicated process, more study needs to be conducted. As each service in microservices has its own database, the process of migration existing database system to microservices requires better tools and knowledge. The migration process should also involve building pipelines for testing purposes, which will create a foundation for further development.

To summarize, this is all not to say Strangler pattern serves no purpose or does not provide enough value; on the contrary, considering medium-size and mid medium size companies, Strangler pattern is a good choice for organizations to migrate into

microservices which achieves both business and development goals. The key is to keep patience, ensuring that migration does not happen overnight.

6 Discussions

Microservices is a system architecture for an application as a collection of services. Each service in microservices is interdependent from each other, has its own database, and is managed by a small, independent team. Microservices has proven to have advantages over monolithic application in terms of deployment, scalability, freedom in development services, and security. However, microservices does come with challenges in managing services, testing, identity access management, debugging, etc. Therefore, microservices is a good fit for medium and large organizations, whereas organizations with limited resources should consider carefully before investing in microservices.

Data collected using the monitoring system during migration process can be deemed trustworthy and objective. Monitoring system is set up by Postman with a full endpoint lifecycle that cover all functionality needed from the application. Although That does not mean that 97% availability of using Strangler pattern in migration process will be reproduced exactly due to differences in environment setup of different projects and the fact that availability does depend on other external factors, but in principle the high availability of the system is reproducible.

This research could be vastly improved by taking into account approaches for database migration into the process. Having database migration with extracting database from monolithic application to microservices, especially dealing with foreign key as well as the relationship of tables, seeing how relational database can transfer to non-relational database, handling real-time database transaction during migration process, would provide a great source of data and enable to view the issue from a quite different and important perspective. Unfortunately, this was not possible in our cause due to resource and time limitations. However, it is highly recommended to include in the process of any research on migration process and would be suggested for anyone willing to research further in this particular topic.

The results of the thesis can be valuable for organizations that are in the process or planning to migrate into microservices. However, the most value to interested parties would be in the process rather than the results. The migration process using Strangler pattern from chapters 5.1 to 5.6 allows the results to serve a wider audience and can also be replicated and used as a basis for organizations to develop migration process from monolithic application to microservices as well as develop new service in microservices.

In itself, this case study provided a great learning opportunity for the student. Although student had some previous knowledge on backend technologies, many discoveries followed after this knowledge was learned and can be applied in further development track. The most important of which probably being knowledge and knowhow of

microservices from theoretical standards and implementation, which can be applied to resolve the pain point of organizations that are tempting to migrate into microservices.

There are a lot of opportunities for further research on the topic of accessibility. Aside from migration process, such topics as monitoring and managing a large amount of services, service discovery to automatically identify endpoints, setting up testing environment as well as end-to-end testing (e.g. building pipeline, combining repositories) or managing security aspect of microservices, could be subjects for further research.

References

- AcmQueue 2020. A Conversation with Werner Vogels. URL: <https://queue.acm.org/detail.cfm?id=1142065>. Accessed: 20 March 2019.
- Arsov, K. 2017. What Are Microservices, Actually?. URL: <https://dzone.com/articles/what-are-microservices-actually>. Accessed: 3 February 2020.
- Aurora, F. 2019. 5G and IoT: How will security change?. URL: <https://blog.f-secure.com/5g-and-iot-how-will-security-change/>. Accessed: 15 April 2019.
- Brown, K. 2017. Apply the Strangler Application pattern to microservices applications. URL: <https://developer.ibm.com/technologies/microservices/articles/cl-strangler-application-pattern-microservices-apps-trs/>. Accessed: 25 April 2020.
- Bruce, M. And Pereira, P.A., 2019. Microservices In Action. Shelter Island, Ny: Manning.
- Burns, B. & Tracey, C., 2018. Managing Kubernetes. First edition edn. Beijing: O'Reilly.
- Burns, B., Villalba, E., Strebel, D. And Evenson, L., 2019. Kubernetes Best Practices. 1 edn. O'Reilly Media, Inc.
- Cohn, M. 2010. Succeeding with Agile. Upper Saddle River, NJ [u.a.]: Addison-Wesley.
- Docker 2020. Docker overview. URL: <https://docs.docker.com/get-started/overview/>. Accessed: 10 February 2020.
- Docker 2020. What is a Container? URL: <https://www.docker.com/resources/what-container>. Accessed: 3 February 2020.
- Evans, E. 2003. Domain-Driven Design: Tackling Complexity in the Heart of Software. 1 edn. Addison-Wesley Professional.
- Fowler, M. 2004. StranglerFigApplication. URL: <https://martinfowler.com/bliki/StranglerFigApplication.html>. Accessed: 20 April 2020.
- Google Cloud, 2020. Deploying a containerized web application. URL: <https://cloud.google.com/kubernetes-engine/docs/tutorials/hello-app>. Accessed: 10 May 2020.

IBM Cloud. What are Microservices?. URL:
<https://www.youtube.com/watch?v=CdBtNQZH8a4>. Accessed: 3 February 2020.

Istio Documentation 2020. Security. URL: <https://istio.io/docs/concepts/security/>.
Accessed: 20 March 2019.

Kwiecień, A. 2019. 10 companies that implemented the microservice architecture and paved the way for others. URL: <https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/>. Accessed: 15 May 2020.

Kong, 2020. 2020 Digital Innovation Benchmark. URL:
https://konghq.com/resources/digital-innovation-benchmark-2020/?utm_source=pressrelease&utm_medium=referral&utm_campaign=2020-innovation-report. Accessed: 15 May 2020.

Kubernetes.io 2020a. Viewing Pods and Nodes. URL:
<https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/>. Accessed: 20 March 2019.

Kubernetes.io 2020b. Connecting Applications with Services. URL:
<https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/#exposing-pods-to-the-cluster>. Accessed: 20 March 2019.

Kubernetes.io 2020c. Install Minikube. URL: <http://kubernetes.io/docs/tasks/tools/install-minikube/>. Accessed: 20 March 2019.

Macaulay, T. 2018. Ten years on: How Netflix completed a historic cloud migration with AWS. URL: <https://www.computerworld.com/article/3427839/ten-years-on-how-netflix-completed-a-historic-cloud-migration-with-aws.html>. Accessed: 3 March 2020.

Michael, M. 2019. Episode 21 | The Cloud: Security Benefits, Risks & Why You Should Use It. URL: <https://blog.f-secure.com/podcast-cloud-security/>. Accessed: 15 April 2019.
Microsoft Azure 2017. Sidecar pattern. URL: <https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar>. Accessed: 20 March 2019.

Mouat, A. 2015. Using docker. First edition edn. Beijing: O'Reilly.

Newman, S. 2018. Building microservices. Sebastopol: O'Reilly Media, INC, USA.

Newman, S. 2019. Monolith to Microservices. 1 edn. Sebastopol: O'Reilly Media.

Nodejs.org, 2020. Dockerizing a Node.js web app. URL:
<https://nodejs.org/fr/docs/guides/nodejs-docker-webapp/>. Accessed: 9 May 2020.

Özgür, T. 2019. Better Logging Approach For Microservices. URL:
<https://medium.com/cstech/better-logging-approach-for-microservices-3cc2c45e7aaa>.
Accessed: 20 March 2019.

Papiernik, M. 2017. How To Use Apache as a Reverse Proxy with mod_proxy on Ubuntu 16.04. URL: https://www.digitalocean.com/community/tutorials/how-to-use-apache-as-a-reverse-proxy-with-mod_proxy-on-ubuntu-16-04. Accessed: 9 May 2020.

Polencic, D. 2019. Hands-on guide: developing and deploying Node.js apps in Kubernetes. URL: <https://learnk8s.io/nodejs-kubernetes-guide>. Accessed: 30 April 2020.

Richardson, C. 2020. Pattern: Database per service. URL:
<https://microservices.io/patterns/data/database-per-service.html>. Accessed: 20 March 2019.

Schaffer, A. 2018. Testing of Microservices. URL:
<https://labs.spotify.com/2018/01/11/testing-of-microservices/>. Accessed: 20 March 2019.

Schenker, G. 2018. Learn Docker - Fundamentals of Docker 18.x. UK: Packt Publishing.
The Kubernetes Authors, 2020. What is Kubernetes?. URL:
<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>. Accessed: 3 February 2019.

Vogels, W. 2018. Modern applications at AWS. URL:
<https://www.allthingsdistributed.com/2019/08/modern-applications-at-aws.html>. Accessed: 20 March 2019.

Walls, S. 2019. Migrating a Monolithic Application to Microservices (Cloud Next '19 UK). URL: https://www.youtube.com/watch?v=_azoxefUs_Y. Accessed: 25 April 2020.

Wong, K. 2020. Object-Oriented Design. 1.3.1 – Coupling and Cohesion. URL:
<https://www.coursera.org/lecture/object-oriented-design/1-3-1-coupling-and-cohesion-q8wGt>

Appendices

Appendix 1. Screenshot from Postman monitor

Figure 23. Postman monitor during migration process (1)



Figure 24. Postman monitor during migration process (2)

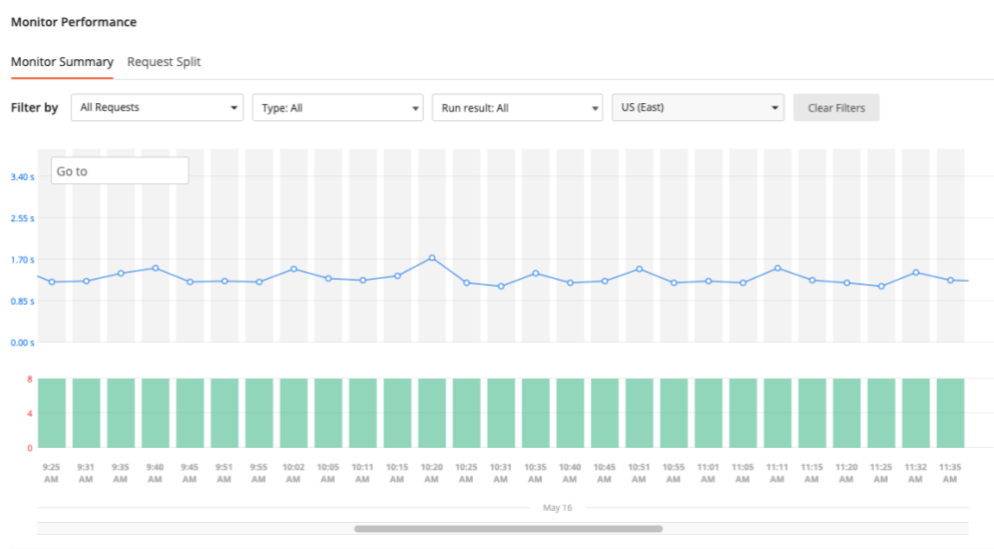
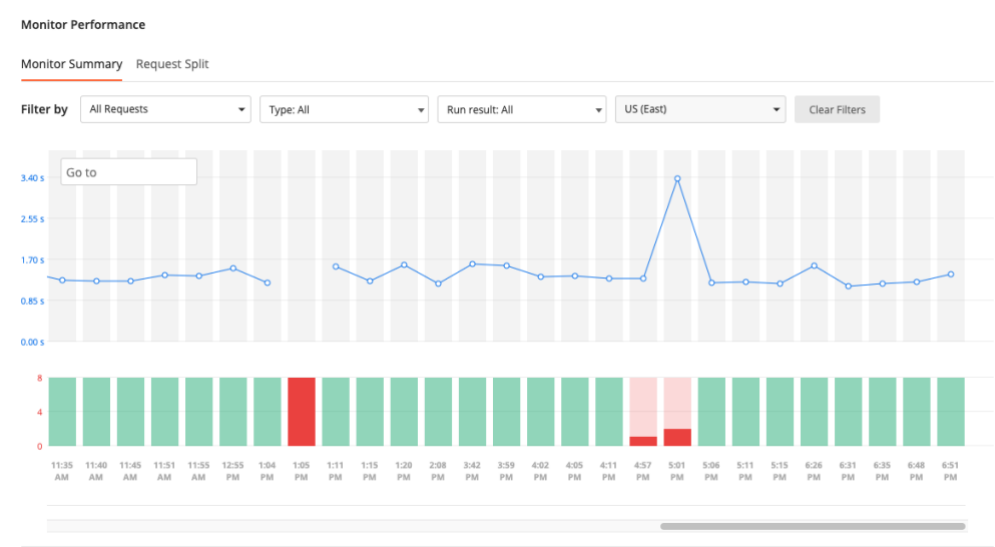


Figure 25. Postman monitor during migration process (3)



Appendix 2. Screenshot of failed attempts' test result

Figure 26. Test results of attempt 1, due to user error

4 failed tests, 2 errors, across 1 region 11:25 PM, 15 May 2020 REGION US (East) [Need help debugging?](#)






















Test Results	Console Log
	 POST Create Baker 35.217.37.57:80/createbaker 200 OK 337 ms 25 B
	 PASS Status code is 200
	 POST Baker Signin 35.217.37.57:80/bakerSignin 200 OK 121 ms 204 B
	 PASS Status code is 200
	 POST Add Product 35.217.37.57:80/addproduct/5ebefaa27403822e95cd675 200 OK 134 ms 396 B
	 PASS Status code is 200
	 POST Search Cake 35.217.37.57:80/search 200 OK 125 ms 805 B
	 PASS Status code is 200
	 PASS Show exact baker email
	 PASS Show exact product id
	 POST Order 35.217.37.57:80/order 500 Internal Server Error 135 ms
	 FAIL Status code is 201
	 DELETE Delete Product 35.217.37.57:80/deleteProduct/5ebefaa27403822e95cd675 503 Service Unavailable 271 ms 377 B
	 FAIL Status code is 200
	 POST Admin signin 35.217.37.57:80/signin 503 Service Unavailable 249 ms 377 B
	 FAIL Status code is 200
	 DELETE Delete Baker 35.217.37.57:80/deleteBaker 503 Service Unavailable 229 ms 377 B
	 FAIL Status code is 200

Figure 27. Test results of attempt 2, due to user error

2 failed tests, 0 errors, across 1 region 11:31 PM, 15 May 2020 REGION US (East) [Need help debugging?](#)






















Test Results	Console Log
	 POST Create Baker 35.217.37.57:80/createbaker 500 Internal Server Error 331 ms 27 B
	 FAIL Status code is 200
	 POST Baker Signin 35.217.37.57:80/bakerSignin 200 OK 253 ms 204 B
	 PASS Status code is 200
	 POST Add Product 35.217.37.57:80/addproduct/5ebefaa27403822e95cd675 200 OK 143 ms 396 B
	 PASS Status code is 200
	 POST Search Cake 35.217.37.57:80/search 200 OK 133 ms 1202 B
	 PASS Status code is 200
	 PASS Show exact baker email
	 FAIL Show exact product id
	 POST Order 35.217.37.57:80/order 201 Created 132 ms 25 B
	 PASS Status code is 201
	 DELETE Delete Product 35.217.37.57:80/deleteProduct/5ebefaa27403822e95cd675 200 OK 132 ms 15 B
	 PASS Status code is 200
	 POST Admin signin 35.217.37.57:80/signin 200 OK 126 ms 268 B
	 PASS Status code is 200
	 DELETE Delete Baker 35.217.37.57:80/deleteBaker 200 OK 131 ms 25 B
	 PASS Status code is 200

Figure 28. Test results of attempt 3, due to user error

2 failed tests, 0 errors, across 1 region 7:24 AM, 16 May 2020

REGIONUS (East)

Test ResultsConsole LogNeed help debugging?


	POST	Create Baker	35.217.37.57:80/createbaker	500 Internal Server Error	542 ms	27 B
	FAIL	Status code is 200				
	POST	Baker Signin	35.217.37.57:80/bakerSignin	200 OK	240 ms	204 B
	PASS	Status code is 200				
	POST	Add Product	35.217.37.57:80/addproduct/5ebf02464771032c6e17a429	200 OK	132 ms	396 B
	PASS	Status code is 200				
	POST	Search Cake	35.217.37.57:80/search	200 OK	124 ms	1202 B
	PASS	Status code is 200				
	PASS	Show exact baker email				
	FAIL	Show exact product id				
	POST	Order	35.217.37.57:80/order	201 Created	129 ms	25 B
	PASS	Status code is 201				
	DELETE	Delete Product	35.217.37.57:80/deleteProduct/5ebf02464771032c6e17a429	200 OK	121 ms	15 B
	PASS	Status code is 200				
	POST	Admin signin	35.217.37.57:80/signin	200 OK	122 ms	268 B
	PASS	Status code is 200				
	DELETE	Delete Baker	35.217.37.57:80/deleteBaker	200 OK	125 ms	25 B
	PASS	Status code is 200				

Figure 29. Test results of attempt 4, due to server configuration

8 failed tests, 16 errors, across 1 region 1:05 PM, 16 May 2020

REGIONUS (East)

Test ResultsConsole LogNeed help debugging?



This run finished without results

Check the console log for details.

Figure 30. Test results of attempt 5, due to library mismatch

1 failed tests, 0 errors, across 1 region 4:57 PM, 16 May 2020 REGION US (East) [Need help debugging?](#)

Test Results Console Log

		POST Create Baker 35.217.37.57:80/createbaker	200 OK 361 ms 25 B
		PASS Status code is 200	
		POST Baker Signin 35.217.37.57:80/bakerSignin	200 OK 131 ms 204 B
		PASS Status code is 200	
		POST Add Product 35.217.37.57:80/addproduct/5ebff15cfb688f6fc93ff83f	200 OK 142 ms 396 B
		PASS Status code is 200	
		POST Search Cake 35.217.37.57:80/search	200 OK 131 ms 805 B
		PASS Status code is 200	
		PASS Show exact baker email	
		PASS Show exact product id	
		POST Order 35.217.37.57:80/order	201 Created 145 ms 25 B
		PASS Status code is 201	
		DELETE Delete Product 35.217.37.57:80/deleteProduct/5ebff15cfb688f6fc93ff83f	200 OK 133 ms 15 B
		PASS Status code is 200	
		POST Admin signin 35.217.37.57:80/signin	200 OK 129 ms 268 B
		PASS Status code is 200	
		DELETE Delete Baker 35.217.37.57:80/deleteBaker	404 Not Found 126 ms 153 B
		FAIL Status code is 200	

Figure 31. Test results of attempt 6, due to library mismatch

2 failed tests, 0 errors, across 1 region 5:02 PM, 16 May 2020 REGION US (East) [Need help debugging?](#)

Test Results Console Log

		POST Create Baker 35.217.37.57:80/createbaker	500 Internal Server Error 526 ms 27 B
		FAIL Status code is 200	
		POST Baker Signin 35.217.37.57:80/bakerSignin	200 OK 265 ms 204 B
		PASS Status code is 200	
		POST Add Product 35.217.37.57:80/addproduct/5ebff15cfb688f6fc93ff83f	200 OK 728 ms 396 B
		PASS Status code is 200	
		POST Search Cake 35.217.37.57:80/search	200 OK 384 ms 1202 B
		PASS Status code is 200	
		PASS Show exact baker email	
		FAIL Show exact product id	
		POST Order 35.217.37.57:80/order	201 Created 419 ms 25 B
		PASS Status code is 201	
		DELETE Delete Product 35.217.37.57:80/deleteProduct/5ebff15cfb688f6fc93ff83f	200 OK 547 ms 15 B
		PASS Status code is 200	
		POST Admin signin 35.217.37.57:80/signin	200 OK 171 ms 268 B
		PASS Status code is 200	
		DELETE Delete Baker 35.217.37.57:80/deleteBaker/5ebff15cfb688f6fc93ff83f	200 OK 332 ms 25 B
		PASS Status code is 200	

Appendix 3. List of abbreviations

GCP – Google Cloud Platform

GKE – Google Kubernetes Engine

JWT – JSON Web Token

DNS – Domain Name Service

IP – Internet Protocol