Tampere University

Lucy Liu

# SMOOTH OPERATIONS FOR LARGE STATEFUL IN-MEMORY DATABASE APPLICATION

## Using Kubernetes orchestration and Apache Helix for improving operations

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
September 2023

# ABSTRACT

Lucy Liu: Smooth operations for large stateful in-memory database application: Using Kubernetes orchestration and Apache Helix for improving operations
M.Sc. Thesis
Tampere University
Master's Degree Programme in Computer Science
September 2023

---

Relex Solutions' Plan product is architecturally a giant stateful monolith with an in-memory database. A system is considered a monolith if all its services need to be deployed together. The database has been kept in-memory because of the data amount the application needs to process and how much faster the performance is when the data is kept in-memory. The Plan architects are looking into taking Kubernetes as an orchestration and lifecycle managing tool. Having an orchestrator in place would provide several benefits, such as automatic scheduling workloads onto a shared pool of resources and better isolation between customers. Kubernetes orchestration is part of bigger architecture initiative to modularize Relex Plan more in attempts to make the monolith more flexible. This thesis is about finding solutions for keeping the operations smooth with Kubernetes and Apache Helix. Literature review and design science will be used as main methodologies for the research.

With Helix role rebalancer and Kubernetes' Statefulset, we can easily scale out and scale in with graceful shutdown. Autoscaling would be well supported by having a resource pool in Kubernetes. Creating pods with Statefulset, make sure each of the pods has a persistent identifier, so rescheduling and restoring pods in Kubernetes native way is covered, while Helix rebalancer takes care that the cluster has wanted number of Plan roles, so there's minimal interruption to the users. Zero downtime would require backwards compatibility for database schema updates, this must be implemented on the product side. The backwards compatibility would technically be a requirement if Kubernetes-native rolling update deployment strategy, with zero downtime, is wanted to take into use in the future. The solution can be applied to other monolithic software architecture with similar setup.

Keywords and terms: Kubernetes, Apache Helix, CI/CD, DevOps, monolith, in-memory application, deployment orchestration, scale-out, scale-in

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

# Contents


1 **Introduction** .......... 1

2 **DevOps by CALMS model** .......... 3

2.1 Fail fast, shift left 5

2.2 CI/CD 6

3 **Cloud Computing** .......... 10

3.1 Service Models 10

3.1.1 IaaS: Infrastructure as a Service 12
3.1.2 PaaS: Platform as a Service 12
3.1.3 SaaS: Software as a Service 13

3.2 Deployment Models 13

3.3 Monolith and Microservice architecture 13

3.3.1 Monolithic architecture 14
3.3.2 Microservice architecture 16

3.4 Containers and containerization 18

4 **Distributed (data) system** .......... 22

4.1 Replication 23

4.1.1 Leader and Follower 23
4.1.2 Synchronous and Asynchronous replication 24

4.2 Partitioning 26

4.3 Distributed Object Storage 27

5 **Tools** .......... 29

5.1 ZooKeeper 29

5.2 Helix 31

5.2.1 AFSM – Augmented Finite State Machine 32
5.2.2 Optimization module 33
5.2.3 Helix execution and the modes 33
5.2.4 Helix roles with ZooKeeper 34

5.3 Kubernetes (K8s/Kube) 36

5.3.1 Basics 37
5.3.2 Pods and Pod scheduling preference 38
5.3.3 Control plane components 39
5.3.4 Node components 40
5.3.5 Service 41
5.3.6 StatefulSet 42

6 **Product Architecture** .......... 43

6.1 Plan Roles and Helix 45

6.2 Distributed commits – component communication 47

6.3 Updates and migration 50

6.4 Motivation for Kubernetes orchestration 51

**7 Research questions and methodology ... 53**

**8 Design suggestions ... 56**

8.1 Adding new instance – Scaling out 56

8.2 Stopping an instance – Scaling in 57

8.3 Restoring instance 60

8.4 Emptying server - Moving instances to another server 62

8.5 Plan application update 66

8.5.1 Adding a column to database schema 67
8.5.2 Deleting a column from database schema 69
8.5.3 Renaming a column in database schema 70
8.5.4 Rollback 71

8.6 Deployment Strategy 72

8.6.1 Rolling update 72
8.6.2 Partitioned rolling update (Canary) 75

**9 Conclusion ... 78**

**References ... 80**

**Appendices ... 88**

# 1 Introduction

Before containerization, cloud and microservices, the traditional software architectures were monolithic. Even nowadays, software might start as monoliths as at the beginning of the project it's easy and simple to develop, test and deploy. At the start of a software business, the requirements and needs might be very simple, a piece of software might fulfil just a single need at the beginning of its lifetime. Thus, it's easy to make a fast profit with monolithic software architecture at the beginning of a project. However, as the business grows and more stakeholders come into the picture, the requirements increase, and different needs need to be considered in the whole infrastructure. Eventually, monoliths might become hard to maintain, especially since they are usually not that flexible, and adopting new technologies and software trends might become very expensive (Atlassian, 2023).

Some well-known software businesses have started as monoliths but eventually remodelled their system into microservices, e.g., Netflix, Amazon, and eBay. For the last decade, microservices seem to have been the trend, however, for some businesses monolithic architecture still makes sense and is even necessary. Microservices might be more popular, but it hasn't totally replaced monolithic architecture.

This thesis is a commission for Relex Solutions company, whom have a product with monolithic architecture with similar characteristics of supercomputing while also taking more cloud computing tools and frameworks into use to match modern needs and demands. These needs not only come directly from customers, but by the company's own goals and motivation to deliver high-quality software. With healthy DevOps culture in place, it naturally raises the need for shifting testing and operations left in the development cycle. Good CI/CD can lead to more robust software and shorter lead time, thus keeping the customers happy while also getting valuable feedback faster. It's important to make sure the feedback loops are short when working with an Agile project. The commissioner's product, which this thesis is focusing on, is slowly moving towards using Kubernetes orchestration. They are interested to know what would be required for the system to run smoothly, without any downtime, in Kubernetes. What would the architectural design look like from operations point of view when they want to keep the existing Apache Helix cluster management. How would Plan environment's lifecycle management work with the Kubernetes orchestrator? How to handle Plan cluster orchestration with the Kubernetes platform?

As part of this work, I will propose solutions on how the product would work with Kubernetes with the requirements the commissioner and the software itself has. Literature review and design science will be used as main methodologies for this research.

Chapters 1-5 of this thesis consist of more theoretical background information on this topic, explaining DevOps and its concepts, cloud computing and distributed data system. Chapter 6 is about the software tools used by the commissioner and the necessary background information of them. Chapter 7 is introducing the commissioner, the product, and its architecture this thesis will be focusing on. Chapters 8-9 include detailed research questions and methods I'm going to use, and the ending design results that have been gone through with commissioner's experts and been approved by them.

## 2 DevOps by CALMS model

The focus in DevOps is not the tooling, but the people. In working environment, company's culture shapes DevOps and gives it its foundation (Freeman, 2019). There are many things that can define a company culture that also shape and characterizes organization.

> Company culture describes the shared values, goals, attitudes and practices that characterize an organization. Aspects such as working environment, company policies and employee behaviour can all contribute to company culture. (Daley, 2022)

Essentially, one could say, that without proper and healthy company culture, good DevOps practices are not achievable. CALMS is an abbreviation of Culture, Automation, Lean, Measurement and Sharing. Figure 1 below illustrates how DevOps is built in CALMS.

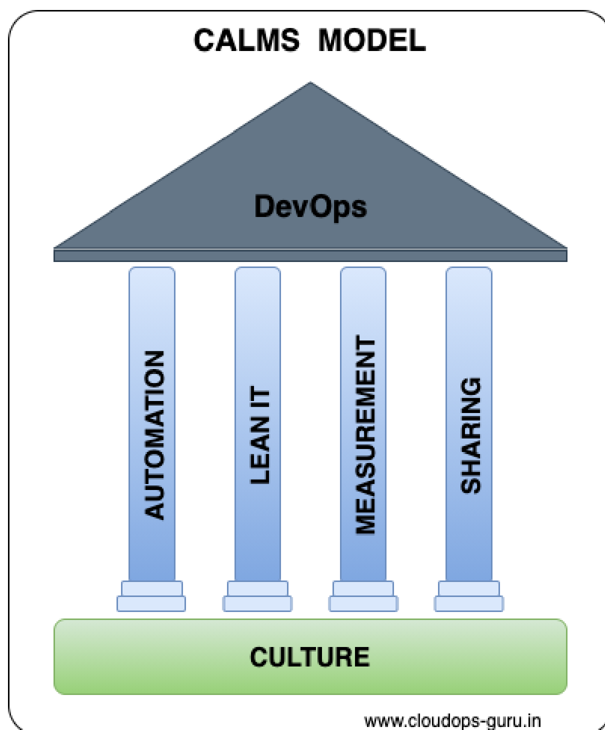



Figure 1. Culture is the very base of DevOps (Nikhil, 2020).

Communication, collaboration, and cross-functionality are central in good DevOps *culture*, as briefly explained previously the whole DevOps came to be because developers and operators were not able to collaborate smoothly. Sociologist Dr. Ron Westrum has defined three categories, shown in below Table 1, of organizational cultures as part of his research "A typology of organizational cultures". *Pathological* culture type describes power-oriented organization where the focus is on personal gain and needs. Rule-oriented

*bureaucratic* types focus on more department level gains. Finally, the performance oriented, *generative* type focuses on collective mission not so much individual, employee or department, gains (Westrum, 2004). The research showed that a company with good information flow optimising culture predicts good outcomes. Google's DevOps Research and Assessment group also recommends companies to follow the generative type of culture (Google Cloud: Cloud Architecture Center, 2023).

| Pathological | Bureaucratic | Generative |
|---|---|---|
| **Power oriented** | **Rule oriented** | **Performance oriented** |
| Low cooperation | Modest cooperation | High cooperation |
| Messengers "shot" | Messengers neglected | Messengers trained |
| Responsibilities shirked | Narrow responsibilities | Risks are shared |
| Bridging discouraged | Bridging tolerated | Bridging encouraged |
| Failure leads to scapegoating | Failure leads to justice | Failure leads to inquiry |
| Novelty crushed | Novelty leads to problems | Novelty implemented |

Table 1. Ron Westrum's typology model shows how organizations process information (Westrum, 2004).

When the Culture, the base part, is covered we can then start to look at the other items in CALMS framework. *Automation* is probably one keyword that pops up right away in people's minds when talking about DevOps. Automation is a good way to tackle repetitive and boring tasks. Automation executes actions more accurately and thoroughly than humans. Continuous Integration and Continuous Delivery (CI/CD) goes often hand in hand with DevOps, but we'll go more into CI/CD later in the chapter. One can list things, actions and items in the development lifecycle that are seen as wasteful actions that don't bring any value to the customer. The idea is to eliminate this waste and that is called the Lean way of working.

In short, *Lean* thinking was invented on Toyota's premises. The principle in Lean is to have "relentless reflection" and continuously improve, the terms used for these are hansei and kaizen, respectfully (Liker, 2016). In order to continuously improve, we have to somehow measure our doings, that way we can also prove that we have actually improved at all. *Measurements* are important in DevOps, key performance indicators should be monitored and there are plenty of tools to help you collect data (Wiedemann, et al., 2019). The bigger question might usually be what to measure, or how the measured data should be analysed. Finally, we come to the *Sharing* part of CALMS framework. DevOps as a community is important, sharing the information and knowledge in the organization, proactively communicating inside a team but also cross-team and even across departments.

## 2.1 Fail fast, shift left

Failing fast is terminology often seen in the agile and lean world. It is quite essential in DevOps lifecycle model, illustrated in Figure 2 below, and also supports the idea of shifting operations left.

> To fail fast means to have a process of starting work on a project, immediately gathering feedback, and then determining whether to continue working on that task or take a different approach—that is, adapt. If a project is not working, it is best to determine that early on in the process rather than waiting until too much money and time has been spent. (Salimi, 2023)

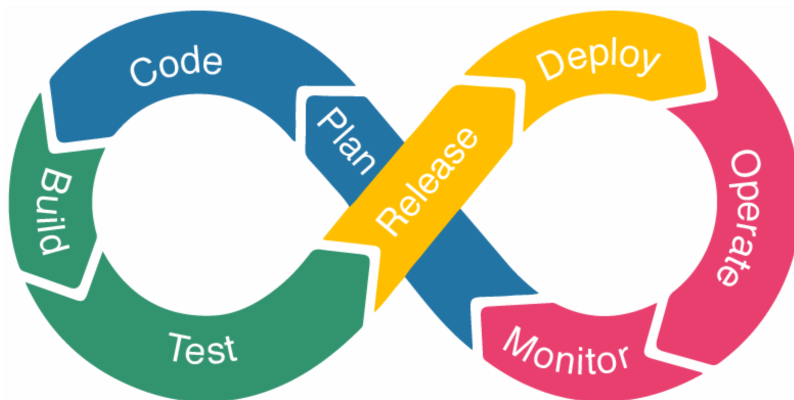



Figure 2. Lifecycle model in DevOps with no clear start and end (Yıldırım, 2019).

According to Freeman (2019); shift left, sometimes also "moving left", the term originates from 1990s testing, to advocate testing as early during development as possible. In the waterfall process (Figure 3), testing happened very late in development which made failures, defects, and bugs very expensive to fix. So essentially, people wanted the testing phase to "shift left", earlier in development. Nowadays the term isn't solely associated with testing anymore, but with other specializations like security and operations as well.

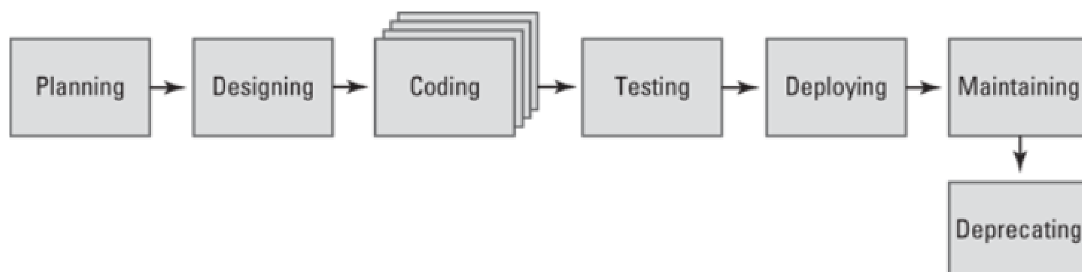



Figure 3. Waterfall development process (Freeman, 2019).

So as said, the main idea in shifting left, is basically moving certain work leftward in the development process. Freeman points out that this philosophy is more about prevention than reaction. We want to prevent potential system failures, not detect them and then try

to fix them. According to a paper released by IBM in 2008, the cost of discovering a defect later in the development process can be up to 30 times more expensive, details in Figure 4 below (Briski, et al., 2008). The DevOps lifecycle model, as presented previously in Figure 2, shows that there's no clear start and end, it's looping indefinitely, which relates to the next Section about continuous integration and delivery.

| Design and architecture | Implementation | Integration testing | Customer beta test | Postproduct release |
|---|---|---|---|---|
| 1X* | 5X | 10X | 15X | 30X |

*X is a normalized unit of cost and can be expressed in terms of person-hours, dollars, etc.
Source: National Institute of Standards and Technology (NIST)†

Figure 4. Cost of defect (Briski, et al., 2008).

## 2.2 CI/CD

Continuous integration and continuous delivery, CI/CD, practise is very much part of DevOps, but as DevOps focuses more on the culture itself, CI/CD focuses on the software development life cycle and advocates automation (Steven, 2018).

In previous Section "Fail fast, shift left", we actually touched a little bit on the CI/CD topic already. It was pointed out that we want to shift operations left, leaving from the classical linear waterfall model as the cost of a defect discovered late in that model is too expensive. Automating CI/CD helps shorten the feedback loops. Mukherjee writes in Atlassian documentation about the business value of continuous delivery, that the automation of continuous delivery helps organizations gain velocity and lower their time-to-market length. Mukherjee however mentions that "speed itself is not a success metric. Without quality, speed is useless" (Mukherjee, 2023). Figure 5 shows the big picture of CI/CD and what they encapsulate. Continuous integration refers to automatic build and test phases. Continuous delivery includes automatic acceptance testing, deployment to staging and smoke testing, while production deployment being manual operation. Continuous deployment would be automating the production deployment as well. In the following sections, I will explain CI/CD in more detail and the value it brings to software development and organization.

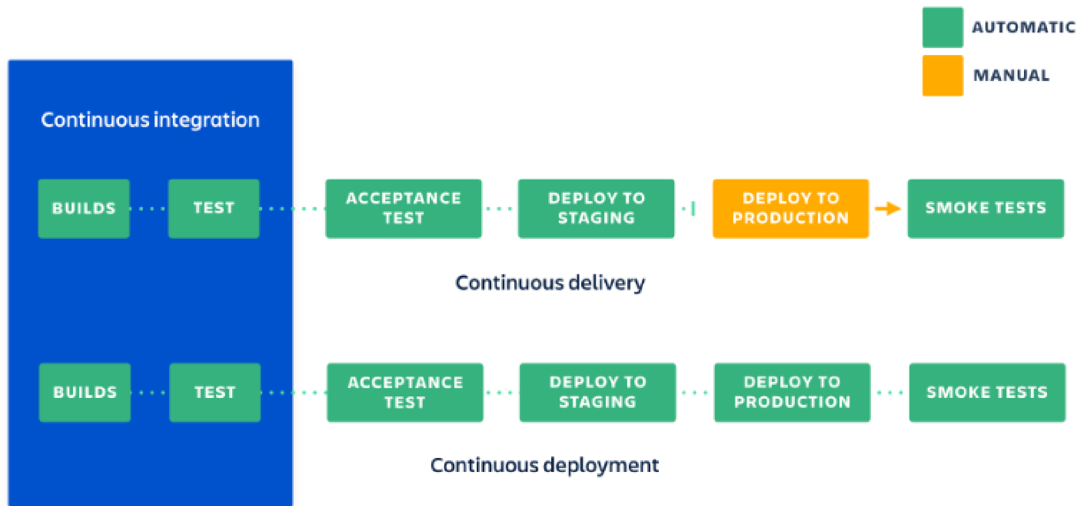



Figure 5. Big picture of CI/CD (Pittet, 2023).

*Continuous integration* is about making sure that the software is always in a deployable state, i.e., the code compiles without errors and the code quality can be assumed to be good as in that the code passes whatever quality gates have been agreed upon beforehand (Rossel, 2017). The integration build might include multiple software components which will be combined into a software system.

With CI we reduce risks by making the software's health measurable which is done by including continuous testing and inspection into the automated integration process. Continuous testing goes hand-in-hand with CI/CD. When CI runs multiple times a day, along with the incorporated testing, the chance of discovering defects when they are introduced is bigger compared to late testing in SDLC (Duvall, et al., 2007).

In software development, there is this "it works on my machine" problem, that the CI essentially also tackles. Basically, developers may have very different kinds of local setups and bunch of environment variables they have set up, which means that one can't really trust that something that works in the colleague's machine would work for everyone else as well. CI build and testing should be done in a clean environment, and should essentially be an independent but repeatable run. Repeatability here means that the same process and scripts are used on a continual basis, this helps reduce assumptions on e.g., the configuration (Duvall, et al., 2007; Belmont, 2018). Continuous integration allows keeping tap on any regressions that the new code might have.

> Continuous Integration can help reduce assumptions on a project by rebuilding software whenever a change occurs in a version control system. (Duvall, et al., 2007)

The automation part, in this context making anything *continuous*, obviously reduces repetitive manual processes that might be human error-prone as well. Developers should be trusting the CI build, as it should establish confidence in the software and make the project more transparent, hence giving more visibility. Developers should be assured that the CI build would essentially catch issues better than when running builds locally (Belmont, 2018).

The CD part in CI/CD is more often regarded as "*Continuous Delivery*" but sometimes also "*Continuous Deployment*". Obviously, deployment and delivery are two different words, so one should know that in software development "Continuous Delivery" and "Continuous Deployment" also mean different things. We should actually be talking about CI/CD/CD instead of just CI/CD, but as it is hard to achieve Continuous Deployment, and sometimes not even wanted, we tend to talk about Continuous Integration and Continuous *Delivery*.

Continuous Delivery means that the artifacts produced by the CI are automatically deployed to the testing and/or staging environment. It is an extension of continuous integration, which would mean that one would have automated testing, integration and now also an automated release process (Pittet, 2023). The release process isn't truly fully automated yet with only continuous delivery. Referring back to Figure 5, the deployment part was still done manually in continuous delivery. As briefly mentioned before, Continuous Deployment also automates the production deployment i.e., all the changes, that have so far passed all the way to staging, are released automatically to customers or production as we tend to say in the software development world. If we look at below Figure 6, continuous delivery encapsulates development, testing and staging, and continuous deployment adds production to it. The figure also gives some details on these different environments.

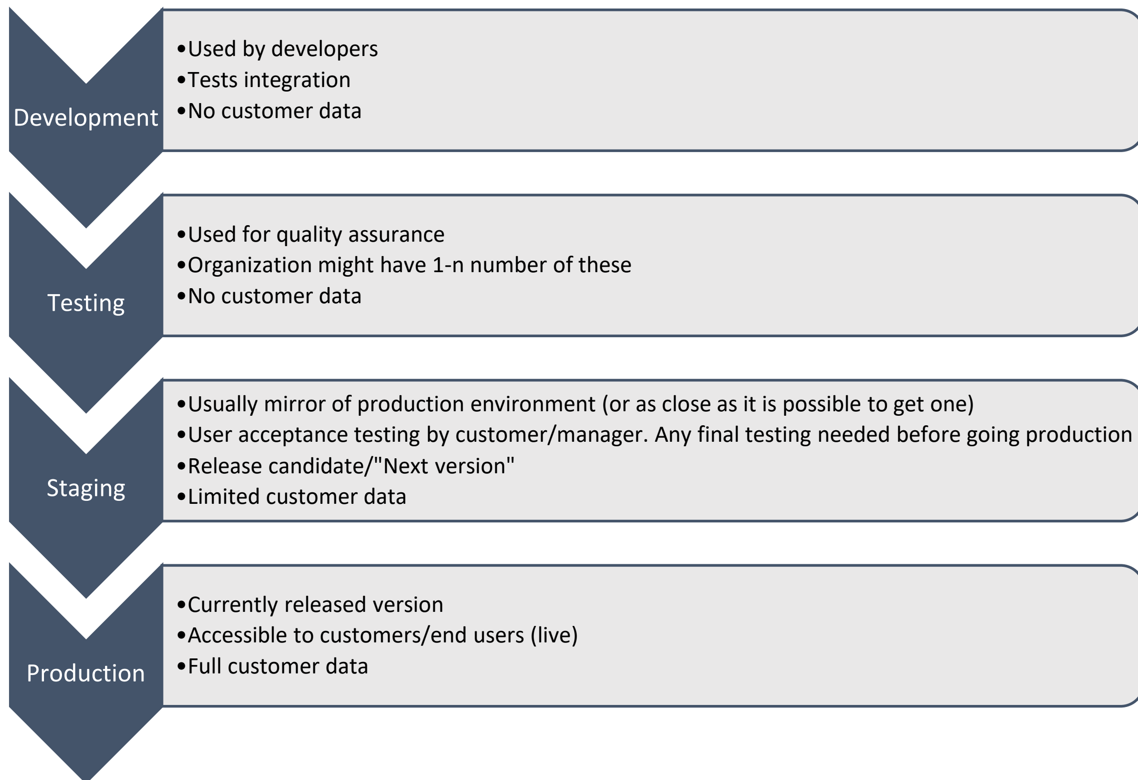



Figure 6. General depiction of different environments in CI/CD.

Likewise, here by automating CD, we minimize risks and reduce manual work. Software deployment becomes easier and the team doesn't have to spend too much time preparing for release activities. The releases can happen more often as well after the complexity of it is taken away. When continuous deployment is part of an automated process as well, the development can also continue without release pauses. Compared to monthly, quarterly or yearly releases, customers could get improvements on a daily basis. The prerequisite is of course that the quality is kept up to standards as well, it's not a good thing to aim for quantity over quality releases. The many stages of continuous deployment are trying to support agile ways of working, with many quality assurance gates. The downside could be the increase in complexity and the bureaucracy that often comes with it. If your product is very small or something like a static or dynamic company webpage, putting complex continuous deployment stages would most likely not bring much value if at all.

# 3 Cloud Computing

Cloud and cloud computing has many definitions and the meaning of cloud has been debated over. The origin of "cloud computing" as a term dates back to 1996, however at that time the meaning was very different as it is now. Back then the term "creators" were looking for how to market their telecommunication business with slogans (Regaldo, 2011). Cloud was a metaphor drawing of internal networking infrastructure with several combined services and technologies. This was surrounded by the customer's facilities. Telecomm companies didn't want to give out too much information about the internal structure or confuse the customers with unnecessary details, hence the cloud (Fogarty, 2012).

For today's definition, instead of me trying to explain cloud computing in my own words, I believe it's better of giving a definition from a trustworthy source. The National Institute of Standards and Technology (NIST) gives the following definition for cloud computing:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models (NIST, 2011).

The five essential characteristics stated by NIST, briefly summarized into key points, are the following:

1. On-demand self-service: Consumer can single-handedly provision resources.
2. Broad network access: Capabilities are available over the network.
3. Resource pooling: Computing resources are pooled and dynamically assigned using a multi-tenant model.
4. Rapid elasticity: Capabilities provisioned and released by demand.
5. Measured service: Resource usage monitoring, controlling and reporting. Automatic optimizing.

## 3.1 Service Models

There are three traditional service models defined by NIST: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Each of these have their own abstraction levels where the lower levels are incorporated in the higher ones,

see Figure 7. The figure illustrates that the lowest abstraction level in these three traditional service models is IaaS, the highest one being SaaS, leaving PaaS in the middle. (Ruparelia, 2016)

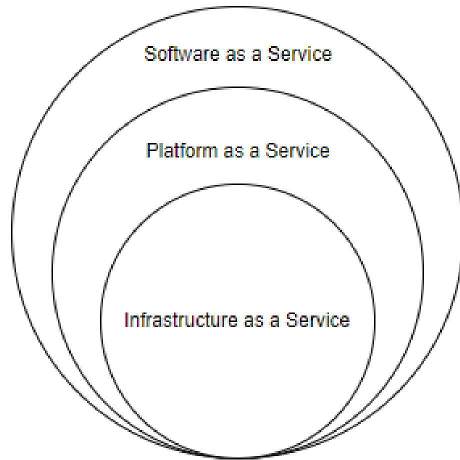



Figure 7. Abstraction levels SaaS > PaaS > IaaS.

The below Figure 8 will help one understand better what is included in each service model. I recommend referring to it when reading the following subsections about each service model presented.

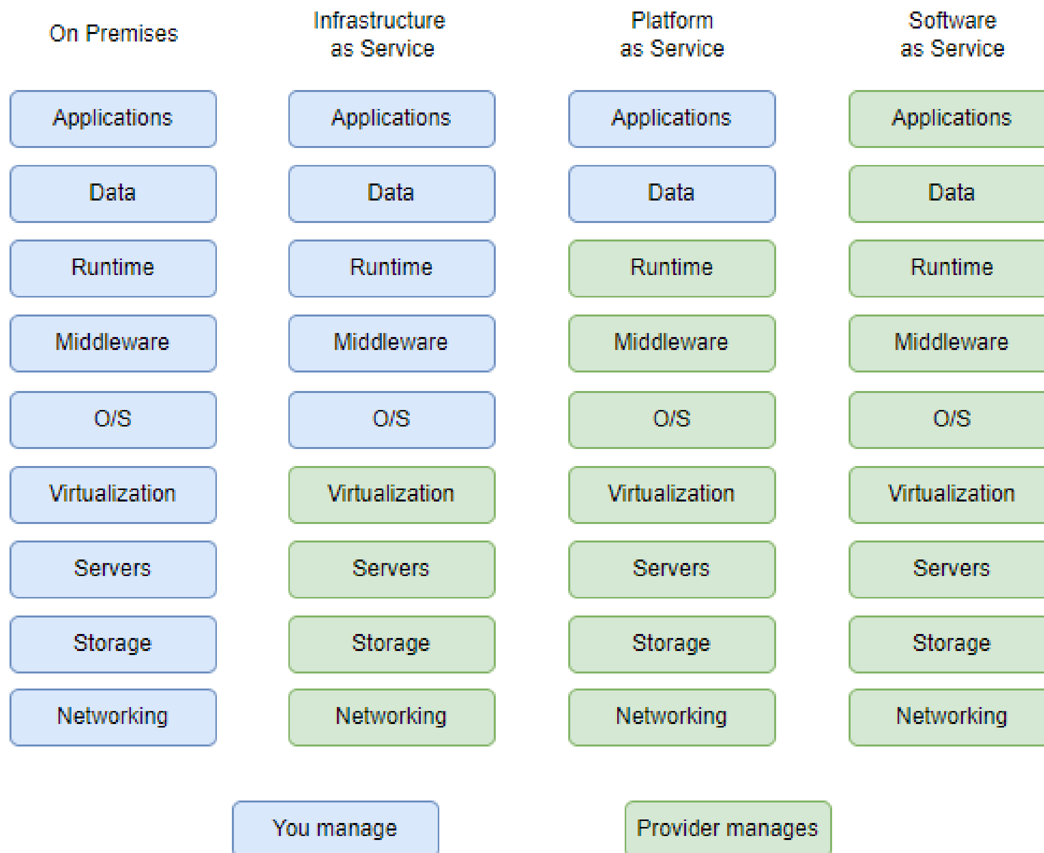



Figure 8. Cloud service model comparison (Kobilinskiy, 2019).

### 3.1.1 IaaS: Infrastructure as a Service

The lowest abstraction level is IaaS, where the customer is provided with hardware (servers and storage), virtualization and/or networking. The customer does not control the actual infrastructure but is allowed to manage operating systems, deployed applications and middleware (NIST, 2011).

IaaS is usually billed by usage. Companies might get IaaS providers in order to expand their capacity, e.g., they can rent datacenters instead of building one themselves. Companies might also need temporal capacity, in which case building one themselves would make even less sense. With IaaS' billing model, customers pay only for what they're using (Roundtree, et al., 2014).

The usage, however, can be billed very differently depending on the provider and their billing model. Companies might be afraid that they are unable to control their usage and how carefully they might need to monitor it. For example, the usage might be measured by resources like processor and memory (Roundtree, et al., 2014). Some examples of IaaS providers are Amazon EC2, Azure Virtual Machine, OpenStack

### 3.1.2 PaaS: Platform as a Service

In addition to what was mentioned in IaaS, PaaS includes the operating system, middleware and runtime to the provided service. The aforementioned are controlled by the provider, the customer has still control over the data itself and applications that are deployed (NIST, 2011).

Having a PaaS helps developers focus on the development work, they don't have to use their time to build, configure or update the servers. Another driver for adopting PaaS is when organizations still want to develop their own application but don't want to take care of the complications of maintaining their own infra and platform, hence deciding on PaaS. Challenges for PaaS might include the lack of flexibility with providers, when one can't find a platform set suitable for their use. Security is also one thing to keep in mind as the service provider might have administrative access to your application through the operative system that they manage (Roundtree, et al., 2014). Examples of PaaS providers are Google App Engine, Microsoft Azure, IBM Cloud Platform.

### 3.1.3 SaaS: Software as a Service

SaaS builds on top of both IaaS and PaaS. With the SaaS model, one could say that everything is handled and comes from the provider and customer gets to use the application that is running on cloud infrastructure. Provider handles everything including the application installation, maintenance and the underlying infrastructure (Ruparelia, 2016; NIST, 2011).

SaaS is a very popular service model as Web-based applications consumption has risen over the years. There are challenges with SaaS as well, such as the latency in the environments when a customer is geographically located far away from the actual application host site. Multitenancy also raises an issue if there's a need to do customization between customers. The security concern is also very apparent in SaaS, the provider has to be careful when handling customer data (Roundtree, et al., 2014). Examples of SaaS applications are Microsoft 365, Zoom, Netflix.

## 3.2 Deployment Models

For deployment models, NIST lists four types. Private cloud, community cloud, public cloud and hybrid cloud. The names of the models describe how they are provisioned and are quite self-explanatory. A private cloud is used exclusively by a single organization. Community cloud is for communities in organizations with shared concerns e.g., security requirements and policy. Public cloud is openly used by the general public. Hybrid cloud composes of two or more cloud infrastructures just mentioned. they're bound together to enable data and application portability (NIST, 2011).

## 3.3 Monolith and Microservice architecture

In today's software architecture, there are both monolithic and microservices. The popular trend favours microservice architecture as it is more scalable and flexible, and goes well with modern software development methodologies (ElGherani, 2022). Examples of companies that started with monolithic architecture but have then implemented microservice architecture are Netflix and eBay. However, monolithic architecture is not disappearing and there are still distinct benefits and drawbacks on both architecture models which I will be explaining more in detail in the following sections. Figure 9 below gives good high level description of what is the difference between monolithic and microservice architecture. Microservice architecture consist smaller functionalities, services. While all monolith's functionality is in one process.

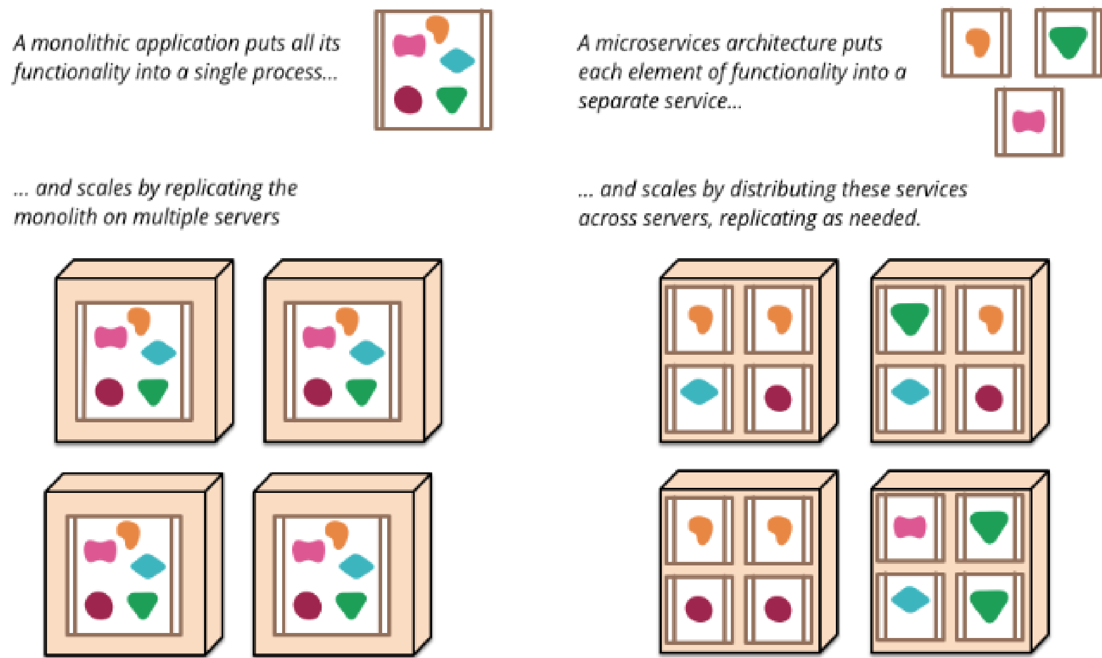



Figure 9. Monolith vs Microservice (Fowler & Lewis, 2014).

### 3.3.1 Monolithic architecture

When searching for a definition of *monolith*, Merriam-Webster gives the following: “a single great stone often in the form of an obelisk or column” and “a massive structure”. The idea is basically the same when we talk about monoliths in software architecture, it’s something that is composed of one single piece, all the modules/components and their functionality is in one enormous application. Essentially, a system is considered a monolith if all its services need to be deployed together.

There are three types of monolithic systems: Single process, modular and distributed monolith. A single process monolith is a system where all the code is deployed as one process. Single process monolith, depicted in Figure 10 in its simplicity, might have several instances behind a load-balancer for horizontal scaling, but it is still considered as single process (Fowler & Lewis, 2014; Newman, 2021).

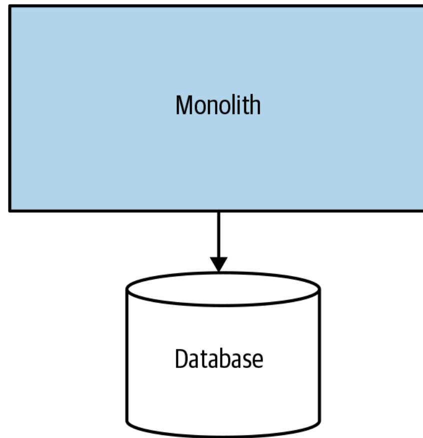



Figure 10. Single-process monolith (Newman, 2021)

Modular monolith, can be considered as subset of single-process monolith. In a modular monolith there are individual modules, each of which can be worked on separately, but for deployment, these modules still have to be combined. Figure 11 below illustrates what modular monolith could look like with single of decomposed database.

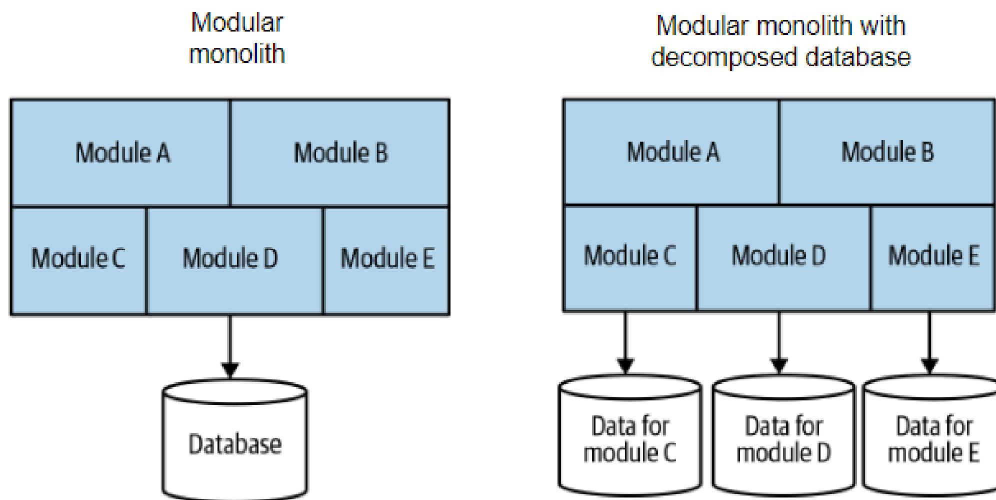



Figure 11. Modular monolith depictions (Newman, 2021).

Then as a third, distributed monolith. A pretty good description of a distributed monolith is that it's basically a microservice, it consists of multiple services, but for some reason, the entire system must still be deployed together. Distributed monolith is not an ideal type, as it comes with downsides of both microservice and single-process monolith without actually bringing enough perks of them. This kind of highly coupled architecture typically appears in cases when information hiding and cohesion of business functionality was not considered enough (Newman, 2021).

The benefits of having a monolith are essentially that it is supposed to be simple. Simple to develop and test end-to-end, easy deployment and horizontal scaling. Monolith also makes code reusing easier compared to distributed system.

Monolithic architecture is most suited for simple systems, when it gets more complex and larger the maintainability also becomes harder and the application is also slower to start up. The entire monolith application has to be basically redeployed on each update, not ideal for continuous delivery. The reliability also is in jeopardy, bugs like memory leaks might affect the whole system. While monoliths can scale quite nicely horizontally, they can basically only scale in that one dimension. It is also hard to keep up with modern technologies and software languages as one would need to consider the whole system (Richardson, 2023).

### 3.3.2 Microservice architecture

Microservice alone could be defined as an application that is independently deployed, scaled and tested. Microservice has single responsibility as it only does one thing. The services should be loosely coupled which means that one service can be changed without touching anything else, this guarantees independent deployability. Microservice architecture is a collection of these individual small applications, services, that then combined build a bigger application (Fowler & Lewis, 2014).

For these individual services to communicate with each other for collaborating and handling requests, they must use interprocess communication (IPC) mechanisms. IPC mechanism can be divided into response-based (synchronous) like REST or gRPC Remote Procedure Call (gRPC) that are based on HTTP, or message-based (asynchronous) protocols like Advanced Message Queuing Protocol (AMQP) or Simple Text Oriented Message Protocol (STOMP) (Richardson, 2023).

Development of microservices is organized around business capabilities which goes very well with DevOps and Agile principles. It advocates having cross-functional teams that work with these business capability-oriented services. The ideal cross-functional team can implement, build, test and operate a service. People would not be grouped based on their core competency aka. functional teams, e.g., Java developers in one team, front-end developers in one and database admins in other. This traditional way of grouping is not bad, but it creates silos which can slow down development and deployment. Cross-functional teams tend to be more efficient and flexible thanks to a wider range of expertise, however, siloed teams might provide a better sense of ownership of something. Individual expertise in cross-functional teams can also become a bottleneck. Business capability

with cross-functional teams reinforces boundaries for service which keeps it independent (Fowler & Lewis, 2014). One can see the similarities of cross functional teams and their designed microservice, or at least the similar design ideology between them.

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. (Conway, 1968)

Microservice, like its name, is supposed to be a relatively small application which makes it easier for developers to understand. The small size also affects positively on the start-up time. As the services are independent, the fault isolation is better. Unlike in monolithic architecture issues like memory leaks won't be fatal to the whole system. Having a microservice architecture also removes the technology stack limitations that monolith architecture has. An entire new tech stack can be chosen with new services, and rewriting the old ones is easier (Richardson, 2023).

While a single service might be easy to understand is more manageable, microservice architecture does bring a lot of complexity as it is a distributed system. The system as a whole is big and complex, requests that span multiple services must be implemented and tested which is not easy. Deployment will also be more complex as the system is now made out of several different services.

The book *Microservice Architecture* (Nadareishvili, et al., 2016) introduces characteristics to be considered when implementing microservice architecture. I won't go into the detailed character definition in this thesis, but encourage interested ones to check the book out. Simply put, the book defines these layered characteristics (architectural phases): modularized, cohesive and systemized, and uses them to create a maturity model (Figure 12). This maturity model categorises benefits according to the phase and goals, either speed or safety. The model shows the impact and priority of benefits as the system's scale and complexity increase and also the activities that should be taken care of during each phase.

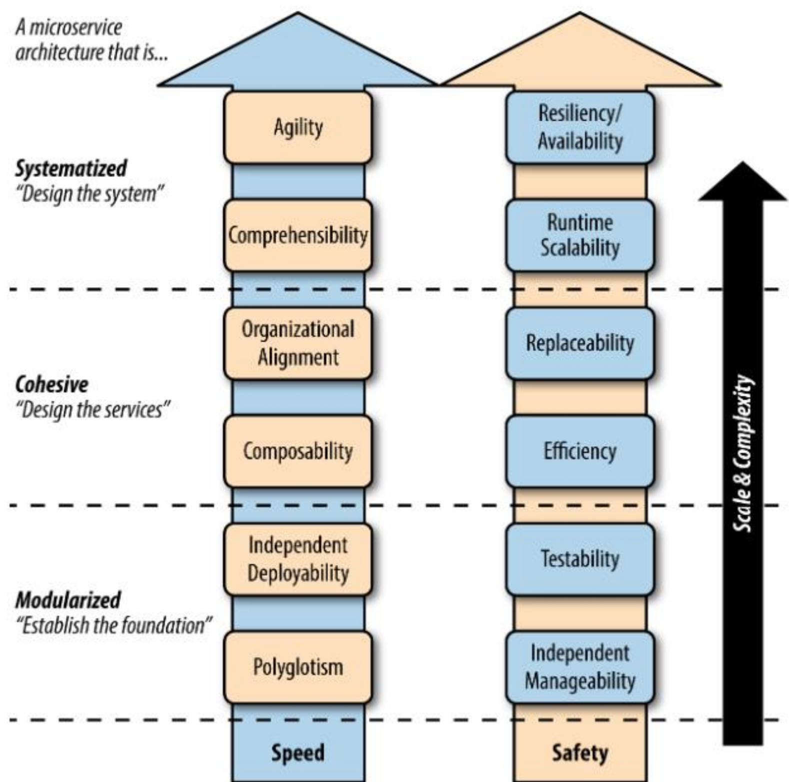



Figure 12. Maturity model for microservice architecture goals and benefits (Nadareishvili, et al., 2016).

### 3.4 Containers and containerization

The container term comes up quickly when developing cloud applications. Containers are standardized packages of computational environments that have the application code and all the dependencies needed to run the software service, such as frameworks and libraries. A container is something that can be easily shared and used. Containers allow us to virtualize both the operating system and the hardware by wrapping them up (Schenker, et al., 2019).

Containers are frequently compared to Virtual Machines. Both technologies allow the isolation of the application from other services and from the underlying hardware. Below Figure 13 compares these two technologies.

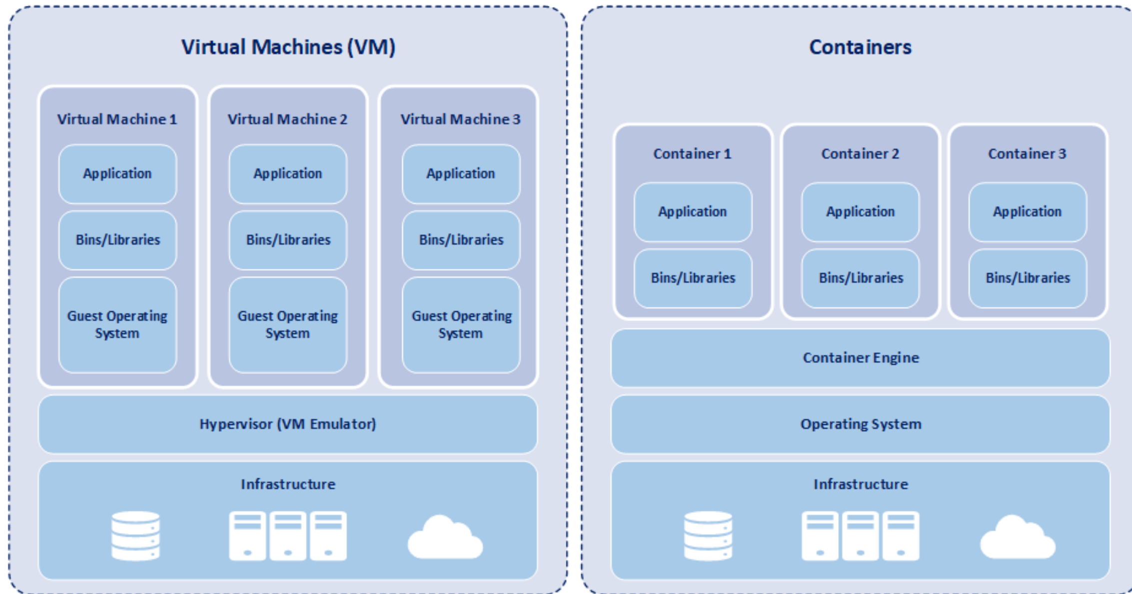



Figure 13. VM vs Containers (Getz, 2021).

Virtualization simulates the physical hardware (CPU, disk and memory) and represents individual machines. Virtualization has its own guest operative system and Kernel, it is known as hardware-level virtualization. Containerization whereas simulates the operative system on the machine, not the entire physical machine. Containerization allows multiple applications to share the same OS kernel (Shejwal, 2022).

Containers make deployments easier by solving two core problems: configuration and infrastructure management. With containers, it is easy to develop and test with a similar configuration as what is running in production. Infrastructure management is made easier as one can run multiple containers in one machine. Organizations don't have to bother themselves by choosing the exact size of a machine, instead larger machine can be purchased for running several containers on it (Barlett, 2023).

*Container managers* are a way to manage the life cycle of a container. The six phases in the container lifecycle are: Acquire, Build, Deliver, Deploy, Run and Maintain. *Acquiring* the base layer container image of the application starts the lifecycle. *Build* phase includes packing in the container image all the necessary application components and libraries, which after the image is published in a public or private repository. During *Delivery* phase, the built application is delivered to production. Delivery phase might include container image vulnerability scanning as well. *Deploy* phase involves of actual deployment of the application to production, and taking care of updates. Management system and runtime environment are set during *Run* step. Any health check, scaling and recovery policies are supposed to be in place for the container after this step. At the end of the lifecycle is *Maintain*, where the application monitoring and any needed maintenance are

taken care of. The system might try to manage failures at run-time for example by container restart. Otherwise, the developers might debug and fix the failure offline and the container lifecycle is re-started again to put out a new containerized version of the application (McGee, 2016; Casalicchio & Iannucci, 2020). Below Figure 14, depicts the container's lifecycle.

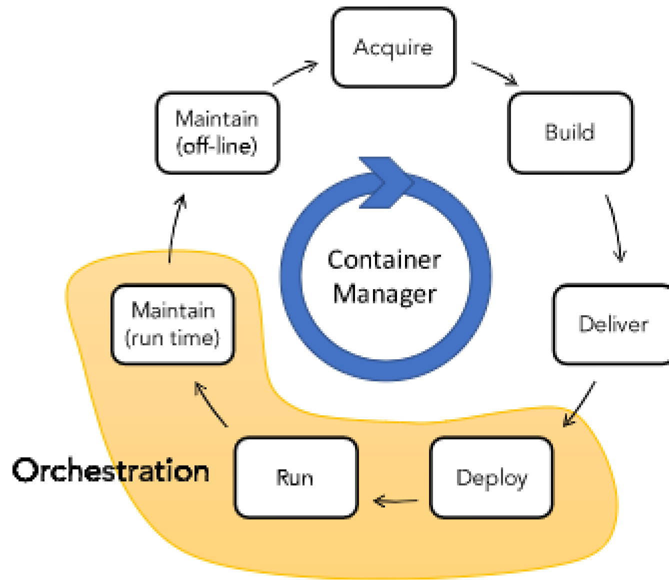



Figure 14. Container lifecycle. Manager provides API to support the lifecycle phases and Orchestration (yellow part) makes it possible to automate certain phases in the lifecycle (Casalicchio & Iannucci, 2020).

*Container orchestration* tools allow automatization of the Deploy, Run and Maintain phases of the container lifecycle, i.e., it automates container task regarding provisioning, resource management and service management. The below Figure 15 lists various functionalities that container orchestration tool can handle for container scheduling, resource and service management.

Figure 15. Orchestration tool can handle scheduling, resource management and service management tasks (Isenbeg, 2017).

The de-facto choice of tools for container or cluster management and orchestration are Docker and Kubernetes, respectively (Docker, 2023; Google Cloud A, 2023). Docker also has its own orchestrator called Docker Swarm. This thesis will mostly focus on Kubernetes and another tool called Apache Helix, which will be introduced properly in the later part.

# 4 Distributed (data) system

There have been several different definitions of what a distributed system is. For this thesis purpose, the loose characterization given in Distributed Systems book (Van Steen & Tanenbaum, 2017) is good enough: “A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system”. The quoted definition includes two features that are characteristic in a distributed system. The first feature is that the system contains independent computing elements (nodes), that can be software processes or hardware devices. The second feature from the definition refers that the users, both people and applications, think that they are using a single system. Other characteristics of a distributed system include resource sharing, simultaneous processing, scalability, easy error detection and node transparency in the means of easy communication between the nodes (Zettler, 2023). With a group of nodes we have clusters, there exists dedicated cluster management software that helps coordinate tasks in the cluster.

The opposite of distributed system is a centralized system, where all computing is done by a single computer in one location. The main difference is that in a centralized system, clients can easily congest the network as all the nodes access the central node where the system state is kept. Basically, a centralized system can fail from a single point. This all might sound familiar with the description of monolithic and microservice architecture in the Cloud computing chapter as cloud computing and microservice are types of distributed systems (Van Steen & Tanenbaum, 2017). Benefits of having distributed system include but are not limited to high availability, scalability and low latency.

*High availability* is valuable to have for organizations disaster recovery plan. High available environment means that there’s minimal service interruption if some member of a cluster malfunctions, otherwise these high available environments should operate without unplanned outages and during specified operating hours (Piedad & Hawkins, 2001). Basically, the other members in the cluster can act as a backup if one or more members come down. A failover process happens, which during the system transfers all the traffic to a member that has been so far redundant or on standby mode. While HA is supposed to have minimal downtime, a step further from that can be thought to be Fault Tolerant, where the goal is to have zero downtime.

*Scalability.* Scaling includes both vertical and horizontal scaling. Vertical scaling, aka. scaling up, basically means that we are scaling to a higher load, buying a more powerful machine is one way to scale up. Scaling horizontally, aka. shared-nothing architecture allows spreading the load of a single machine to multiple machines. If a single server gets

overloaded with requests it will become a bottleneck and affect the performance (Van Steen & Tanenbaum, 2017).

*Low latency* rate is important for users and businesses. Organization can have servers at various locations so users can be served from the geographically closest datacenter – this allows network packages to travel faster to the client (Kleppman, 2017).

In the following sections, I will briefly explain more about replication, partitioning and object storage. The concept of replication and partitioning is good to comprehend for understanding some of the tools and architecture I'll introduce later. In replication and partitioning, a database is used as an example, but the concept itself is not exclusive to data distribution.

## 4.1 Replication

Replicating data in a distributed system means that we are copying the same data to multiple machines connected via a network. Replicating helps reduce latency and increases both availability and read throughput, the last one meaning that more nodes/machines can serve read queries (Kleppman, 2017).

### 4.1.1 Leader and Follower

Nodes that store a copy of the database are called replicas. The leader or primary replica is considered to be the one which receives the requests from the client to write new data. The new data will be written to the local storage of the leader replica. The leader will send the data change to all the other replicas, aka. followers or secondaries, which will apply the changes in their own local copy. Both leader and follower can be read but only leaders accept writes from clients (Kleppman, 2017). This is illustrated in the following Figure 16.

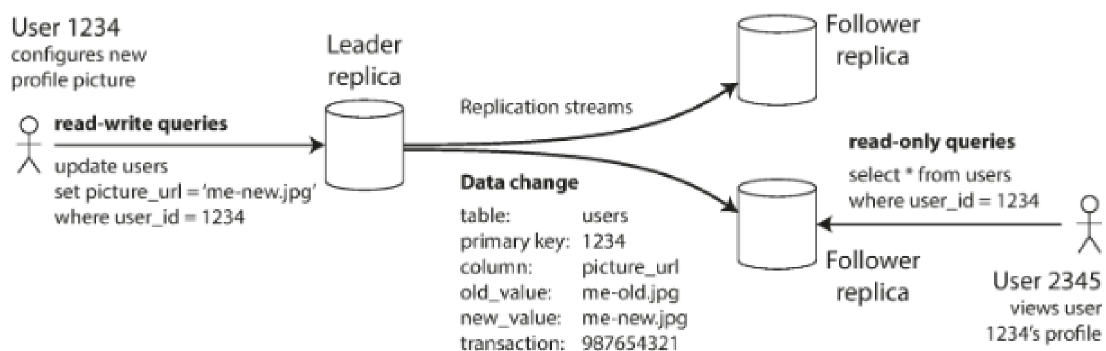



Figure 16. Leader-Follower replication (Kleppman, 2017).

### 4.1.2 Synchronous and Asynchronous replication

Replication can happen synchronously or asynchronously. Figure 17 shows in sequence diagram how synchronous replication happens. The leader in the synchronous replication will wait to receive confirmation from both of its followers until it will report success to the user. Whereas in asynchronous replication, Figure 18, the leader doesn't wait for the follower's confirmation.

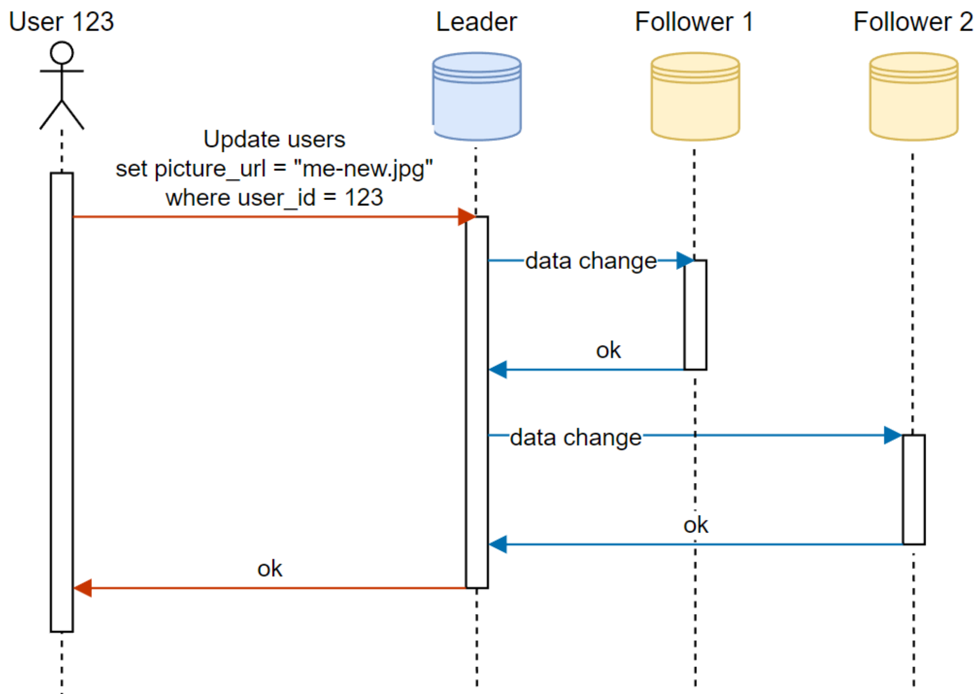



Figure 17. Leader-based synchronous replication. Adaption of (Kleppman, 2017).

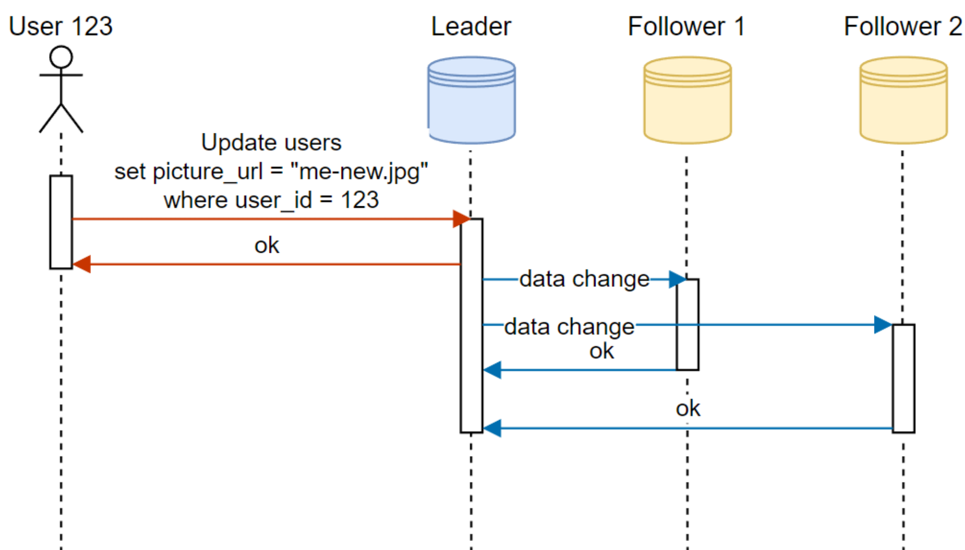



Figure 18. Leader-based asynchronous replication. Adaption of (Kleppman, 2017).

Not all the followers have to use one type of replication, unlike in the diagrams above, there can be a mix of asynchronous and synchronous followers, and this is the advised way. Synchronous replication makes sure that the follower's data matches with the leader's data. If the leader replica were to crash for any reason, the followers would still have the up-to-date data as backup. The downside of the synchronous replication is that if one of the followers has crashed, the writing process will be stuck. Semi-synchronous replication, Figure 19, can help to migrate such a problem.

In semi-synchronous replication, one of the followers is synchronous and the rest are asynchronous. A system relying solely on synchronous replication wouldn't really work in practice, as one follower failing would halt the whole system. With a semi-synchronous configuration, the async follower will be made to synchronous if the original synchronous follower becomes unavailable (Yoshinori, 2014).

Fully asynchronous configuration is also possible, but it is considered as less robust as the writes that haven't been processed by the followers will be lost if the leader fails. Of course, the flip side is that leader would never be in a halt even if all its followers fail (Kleppman, 2017). If a leader replica goes down for some reason, a similar failover process happens that was mentioned in the high availability section earlier. Basically, some other follower replica will be promoted to take the leader role.

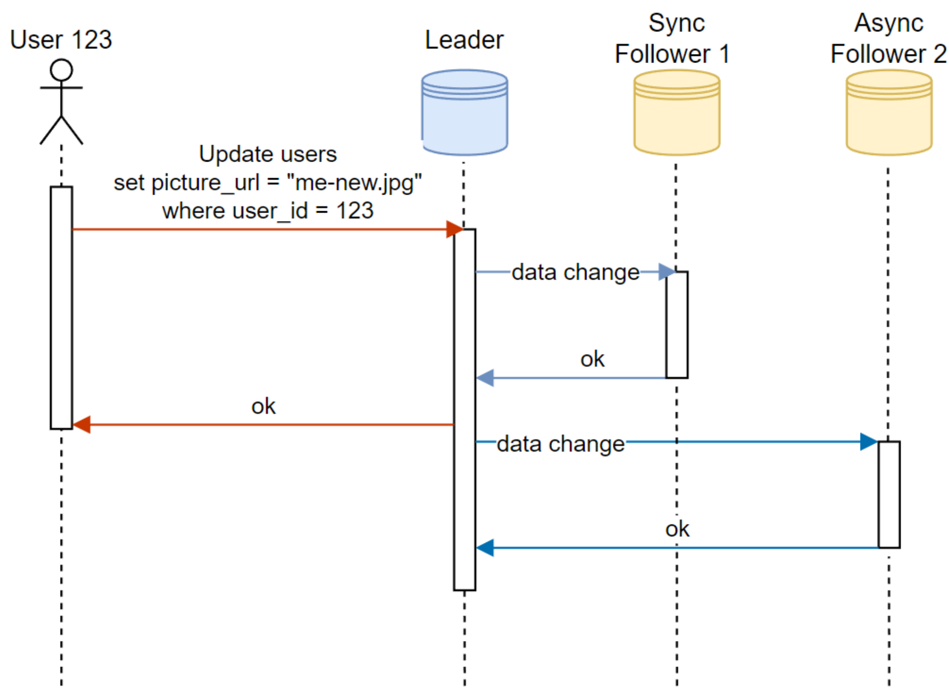



Figure 19. Semi-synchronous replication.

## 4.2 Partitioning

Simply replicating one copy of the same data to multiple different nodes might not be the ideal nor sufficient solution with very large datasets. In such cases, partitioning – breaking data into smaller parts, comes handy. Partitioning supports scaling out better, with the goal of spreading data and query load to different nodes. Each partition can be considered as a small database on its own, but the database can support operations that affect multiple partitions. A piece of data usually belongs only to one partition. A big dataset can be distributed across many disks when we place different partitions on different nodes, which allows distributing the query load to multiple processors (Kleppman, 2017).

Everything mentioned in the replication section applies to partitioning. For fault tolerance, we might want to store multiple replicas of partitions in different nodes. A single node can store more than one partition also. Figure 20 illustrates an example of how would partitioning and replication work together in the leader-follower model. The figure shows how each node has one leader of a certain partition and followers of other partitions, the followers of the same leader are spread on the other nodes (Kleppman, 2017).

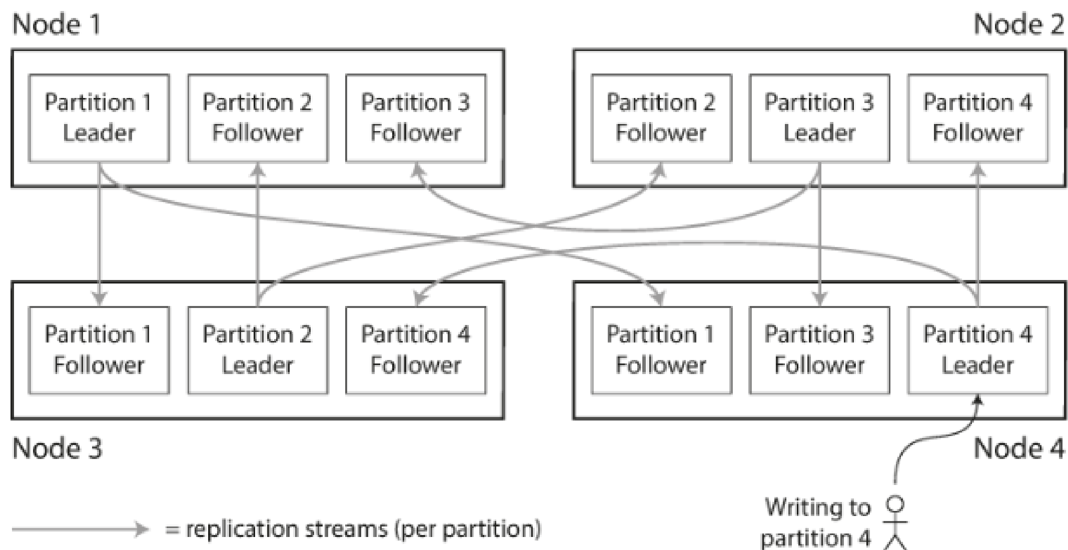



Figure 20. Partitioning and replication (Kleppman, 2017).

When the data is partitioned to different nodes, how does a client know which node it's supposed to connect to when making a request? How does the service discovery happen?

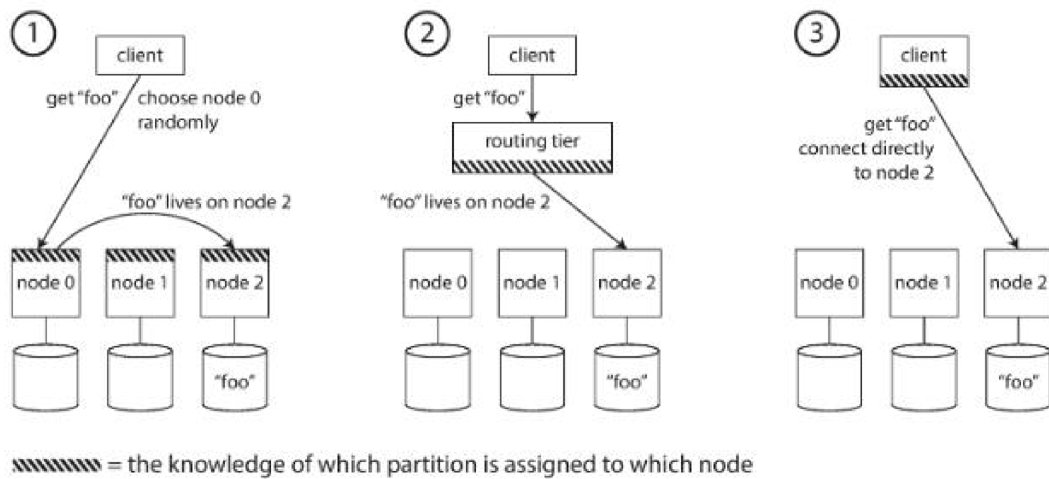



Figure 21. Ways to route request to the correct node (Kleppman, 2017).

The above Figure 21 shows some examples of how to make the service discovery happen:
1. Client connects to a random node which will either handle the request directly if it owns the partition or forwards the request to the correct node.
2. Routing tier that has the information about the partitions. It will forward all the requests from the client to the correct node.
3. Clients know already which partition is assigned to which node so they can connect directly.
Essentially there is still the problem of how does the routing decision-making component know about the changes that might happen in the partitions to node assignment? The changes are caused by manual or automatic rebalancing, which essentially happens when e.g., more CPU, RAM or disks are added. Some services can be used to keep track of the cluster metadata (Kleppman, 2017). I will introduce one of those services, Apache ZooKeeper, in the later chapter.

### 4.3 Distributed Object Storage

File storage is a more traditional type of storing data. The data is stored in a hierarchical structure, with directories and folders. One would need a correct path to find a piece of data. File storage is good for storing small amounts of data and making it easily accessible to multiple users. The third type of storing is block storage, where files are broken and stored as separate blocks. Unlike file storage, block storage doesn't use single path to data which benefits the performance. Block storage can be good when e.g., handling a large amount of transactional data, but it can be expensive. (Google Cloud B, 2023).

Object storage is an architecture model of data storage for storing unstructured data in a format called objects. The data is perceived as units that also include metadata and unique

identified used to locate and access the said unit. These units are placed in a storage pool, a flat data environment which can be scaled easily by adding more storage devices to increase the size of the pool. The objects can be easily located and accessed through REST API. In addition to the mentioned benefits, object storage provides good data durability and resiliency as the data can be replicated and stored across several devices and geographical regions. Nowadays distributed storages are usually hosted on cloud services. The downside of object storage is that it's not good for constantly changing dynamic data, as one would need to rewrite the entire object when the data changes (Google Cloud B, 2023). Examples of object storage services are Amazon S3 and Azure Blob Storage.

# 5 Tools

This chapter will explain in high detail cluster management and orchestration tools, Zookeeper, Helix and Kubernetes, that are related to this thesis. These tools are used by the commissioner which is why it is good to understand the basics before jumping into the commissioner's architecture in later chapters.

## 5.1 ZooKeeper

I mentioned very briefly about ZooKeeper in an earlier section saying that it's a tool for managing cluster metadata. Here's a more detailed definition given on ZooKeeper's home page: "ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services", (Apache ZooKeeper, 2023). ZooKeeper acts as a coordination service, to which nodes will register themselves to. The mapping of partitions to nodes is maintained by the tool, and other services/actors that need this information can subscribe to it. ZooKeeper will then notify the subscribers whenever a node is added or removed or if ownership of the partition is changed (Kleppman, 2017). Below Figure 22 illustrates how ZooKeeper works on a high level in the context of service discovery.

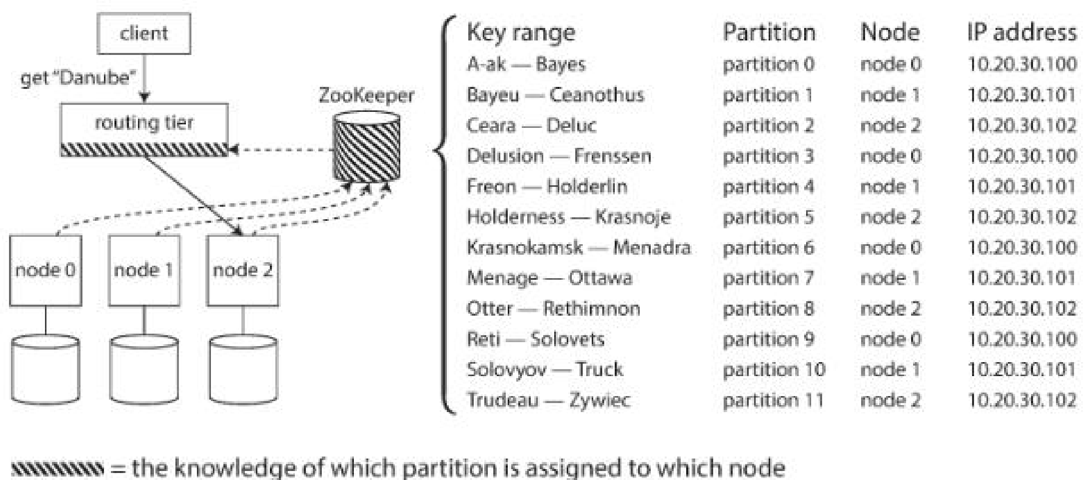



Figure 22. Zookeeper keeps track of partitions and nodes (Kleppman, 2017).

ZooKeeper works on a leader-follower basis, which was previously described in Subsection 4.1.1. Clients can connect to any node for reading data, but writing happens only through leader nodes. ZooKeeper uses a shared hierarchal namespace organized like a regular file system to coordinate distributed processes. Data registers called znodes, similar to files and directories, reside in the namespace. High throughput and low latency can be achieved with ZooKeeper as its data is kept in-memory. ZooKeeper itself is actually meant to be replicated over multiple servers/hosts just like the distributed processes

ZooKeeper coordinates. A cluster of ZooKeeper nodes are called an ensemble. The updates to ZooKeeper are ordered with stamps that reflect the overall order of the transactions (ZooKeeper, 2022).

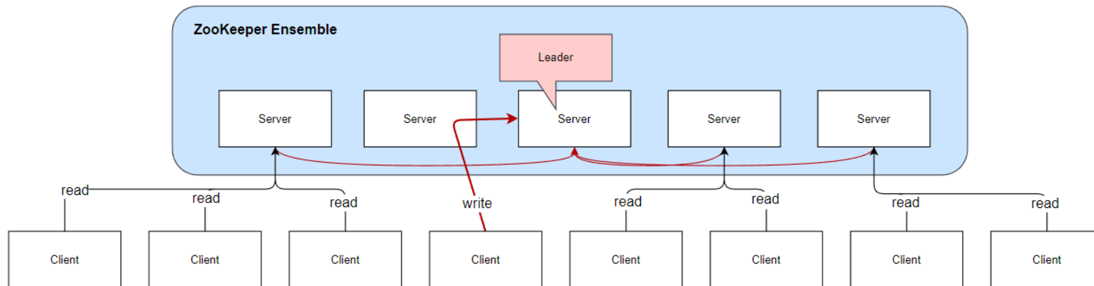



Figure 23. Zookeeper servers share state, logs and snapshots. Adaption from (ZooKeeper, 2022).

Each Znode stores some data which may or may not have children associated with it. The metadata is maintained as a stat structure and these data include version number which is increased whenever some data is updated, access control list and timestamp. ZooKeeper uses an atomic broadcast system which keeps total ordering, meaning that the order of the sent transactions is the same across servers i.e., local replicas never diverge. Atomicity also means that changes are treated as a single operation, they are grouped and processed together, i.e., a group of write operations must all be committed or all rejected/rolled back, regardless of failures (Kleppman, 2017).

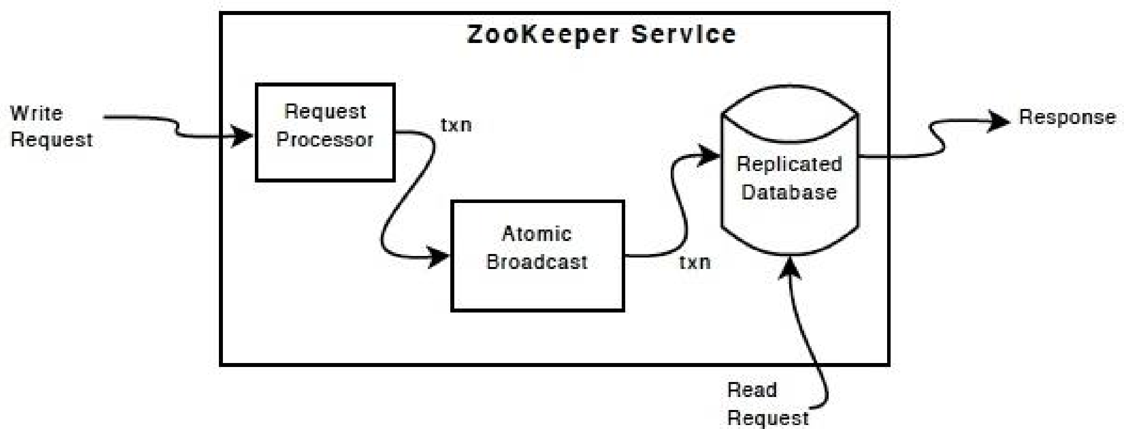



Figure 24. ZooKeeper service components (ZooKeeper, 2022).

Clients can get notified of znode changes with ZooKeeper *Watchers*. To avoid the antipattern of polling events continuously, ZooKeeper has its own service mechanism where clients can get notifications from. Instead of a poll/pull model, Zookep Watcher implements a push model, where registered clients get notifications "pushed to them" when something changes in that particular znode. Watcher is one-time use, so a new watcher has to be set to the znode after the previous one has been triggered. Figure 25 below is an

example where Client1 is interested to get a notification when another node joins the cluster. An ephemeral node (deleted after the creator's session ends) will be created to ZooKeeper path or members when a new node joins. Client2 joins and creates this ephemeral node called Host2 which triggers the watcher. Client1 sets new similar watcher after Client2 has joined (Haloi, 2015).

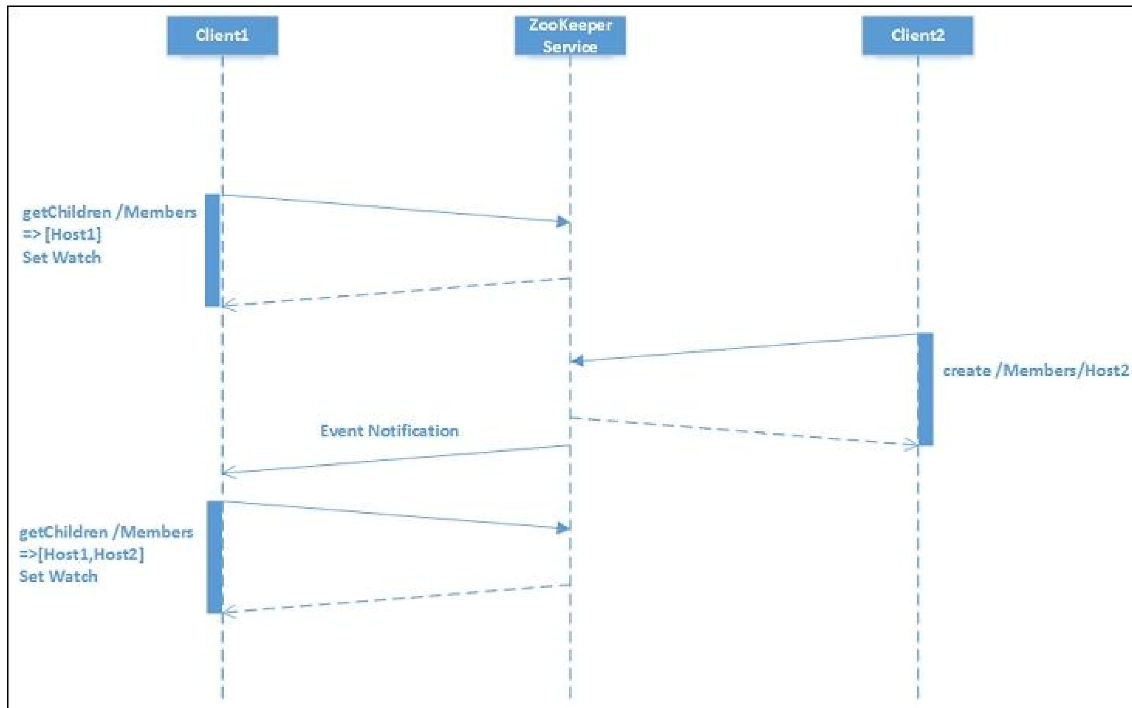



Figure 25. Sequence example of two clients and ZooKeeper with watches and notifications (Haloi, 2015).

## 5.2 Helix

Apache Helix is a cluster management framework; it allows automatic management of distributed resources hosted on nodes within a cluster. From Apache Helix's own home page: "Helix automates reassignment of resources in the face of node failure and recovery, cluster expansion, and reconfiguration". The nodes inside of a cluster are independent and mostly isolated. These nodes can be made aware of other services with communication mechanisms, but they wouldn't need to know about the whole system. In order to manage certain tasks and actions easily, a manager tool can be introduced.

Apache Helix was developed at LinkedIn. Helix is a generic cluster management system that abstracts common cluster management tasks. LinkedIn people defined the following common tasks for running and maintaining distributed data system (DDS): Resource

management, fault tolerance, elasticity and monitoring. See the following Table 2 for more detailed definition of each mentioned task.

| Task | Definition |
|---|---|
| **Resource management** | DDS provided resource (database, index, etc.) must be divided between cluster's nodes. |
| **Fault tolerance** | Node failure doesn't crash the whole DDS, the system shouldn't lose data and read and write operations should still be available. |
| **Elasticity** | Cluster grows and adds more nodes according to the demand and growing of workloads. The resources of DDS get redistributed accordingly to the new nodes. |
| **Monitoring** | Cluster monitoring gives important health metrics and helps noticing fault tolerance problems. Follow-up actions are required afterwards, e.g., lost data re-replication or data re-balancing across nodes. |

Table 2 Common tasks for running and maintaining DDS, defined by experts in LinkedIn (2012).

Helix provides set of pluggable interfaces for declaring DDS's correct behaviour. According to the conference publication "Untangling Cluster Management with Helix" (Gopalakrishna, et al., 2012) the key of the design of pluggable interfaces is augmented finite state machine (AFSM) and the optimization module.

### 5.2.1 AFSM – Augmented Finite State Machine

The DDS uses AFSM to encode the possible valid states, legal transitions and related constraints that it can have. A good real-life example of a finite state is a traffic light. A traffic light has states of red, yellow, and green, and the transitions are red-yellow-green and green-yellow-red. An example of a constraint could be the pedestrian traffic light, which shows red until someone pushes the pedestrian button. Another example of constraint, closer to the topic, is that we define that a partition can only have one leader. Helix's AFSM works on partition level (Gopalakrishna, et al., 2012).

### 5.2.2 Optimization module

The DDS can also use an optimization module to specify goals for ideal resource distribution in the cluster. Unlike AFSM, the optimization module can optimize on partition, node, resource, and cluster levels. The optimization can be of two types: transition goal or placement goal. The emphasis is on the goal word, as Helix tries to achieve its goals but not by ignoring the cluster correctness. The cluster's correctness is not dependent on transition or placement goals (Gopalakrishna, et al., 2012).

With transition goals, the DDS is able to tell Helix how to prioritize multiple replica transitions. Helix maintains the correctness of DDS during transitions and throttling by choosing the ordering of the transitions. Basically, Helix has a transition preference list. Placement goals let Helix know how should it place replicas on nodes as there're multiple choices. Load balancing can be achieved for example with the placement goals (Gopalakrishna, et al., 2012).

### 5.2.3 Helix execution and the modes

I have now described how DDSs behaviour is declared in Helix with AFSM and an optimization module. Helix monitors continuously the DDS state and orders transitions on it whenever it's necessary. This way, it makes sure that the declared behaviour is being followed during run-time.

```
 1: repeat
 2:    validTrans = ∅
 3:    inflightTrans = ∅
 4:    for each partition p_i do
 5:       Read currentState
 6:       Compute targetState
 7:       Read p_i pendingTrans
 8:       inflightTrans.addAll(pendingTrans)
 9:       requiredTrans = computeTrans(currentState, tar-
          getState, pendingTrans)
10:       validTrans.addAll(getValidTransSet(
          requiredTrans))
11:    end for
12:    newTrans = throttleTrans(inflightTrans, validTrans)
13: until newTrans == ∅
```

Figure 26. Helix's execution algorithm (Gopalakrishna, et al., 2012).

According to the LinkedIn people (Gopalakrishna, et al., 2012), the key feature of the Helix transition algorithm is that the same algorithm can be applied to all different changes and across all DDSs. The conference paper shows the algorithm, Figure 26, and unravels the algorithm in detail, which I recommend reading from the paper itself if one

wants to have deep understanding. The algorithm reads the current state of DDS and computes its target state. The current state will most of the time already match with the target state and nothing will happen. Only if the cluster changes for some reason (e.g. adding or removing partitions) will the state be different. Helix uses "Controller, Scalable, Decentralized Placement of Replicated Data"-algorithm, abbreviated as CRUSH. It is an improvement of the RUSH algorithm and helps Helix achieve load-balancing goals. In a nutshell, CRUSH distributes data according to a weighted hierarchy on available storage transitions and then according to the requirement, adds additional replica transitions (Weil, et al., 2006). The algorithm takes account of pending transitions and computes a set of valid transitions accordingly. These valid transitions are computed for maximizing the number of transitions that can be done in parallel. Helix will add transitions to the set according to the priority order declared by DDS in the optimization goals. Using the CRUSH algorithm Helix will fill the set while taking throttling optimization goals into account. When the transitions are completed, the call-back notifies Helix to remove the transitions from pending transitions in their respective partitions list (Gopalakrishna, et al., 2012).

Helix has three different modes so the users are not forced to use the default CRUSH algorithm but instead, it's possible to give application the control of placement and the state of the replica. The default mode of execution is AUTO, which as mentioned, uses the CRUSH algorithm. With AUTO mode, the state and placement of the replica is decided solely by Helix. The conference paper mentions that this mode is mostly used for applications where replica creations is inexpensive. SEMI-AUTO is the second mode of Helix, here the application gets to decide the placement of the replica, but Helix is still taking care of the states of the said replicas. This mode is used when replica creation is expensive which is common with DDSs that have a lot of data. The third mode offered is CUSTOM, which allows DDS to take full control of the placement and state of each replica. Helix coordinates the DDSs state moving from the current to the final state defined by the DDS itself. The application uses an interface provided by Helix to offer custom functionality to handle cluster changes. The CUSTOM mode is used in cases when it's necessary for multiple resource coordination or when additional logic is used to decide the replica's final state (Gopalakrishna, et al., 2012).

### 5.2.4 Helix roles with ZooKeeper

Remembering Figure 22 in the ZooKeeper section, Helix takes the role of the routing tier in the image, using ZooKeeper to detect any changes in the DDS states. Helix has three roles which communicate with each other using ZooKeeper: participant, spectator and

controller. The roles help Helix categorize the nodes logically according to their functionality. *Participants* are nodes hosting the distributed resources. The Participant's current state is observed by *Spectator* nodes, which also do the request routing. Hence, the Spectators need to know the information of the partition's host instance and its state to route the request to the correct endpoint. The *Controller* is a node that both observes and controls Participant nodes. Controller is Helix's core node, which does the transition coordination in the cluster, the finite state machine is hosted in the controller and the execution algorithm is also run in it (Apache Helix, 2023; Gopalakrishna, et al., 2012).

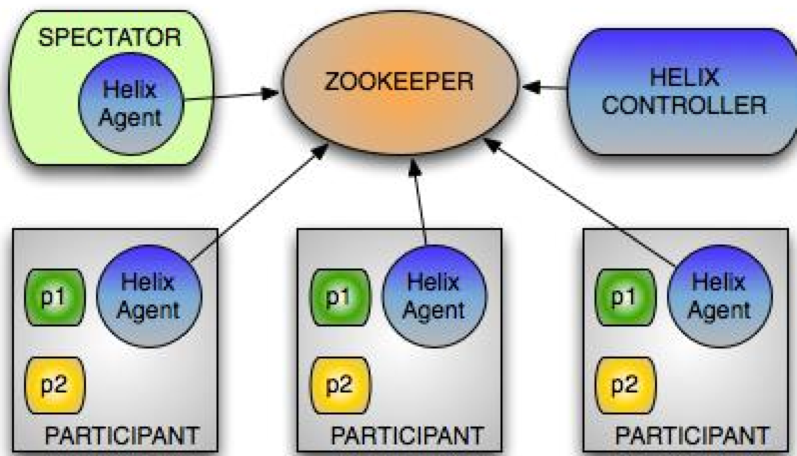



Figure 27. Helix Roles with ZooKeeper (Apache Helix, 2023).

ZooKeeper in Helix provides a communication channel between the controller and spectators. This channel is perceived as a queue in Zookeeper and the producers and consumers on this queue are controllers and participants. "Producers can send multiple messages through the queue and consumers can process the messages in parallel" (Gopalakrishna, et al., 2012). Figure 28 below illustrates the interaction between these three Helix roles. After participants come alive, they wait for new messages in the message queue. ZooKeeper maintains the partition replicas' current and target/ideal states. Helix provides External view for spectators that is a combined view of all nodes' current state. The Participants get sent messages by the controller when it gets notified about the difference in the states, the message indicates the required state transition as a task which the Participant will perform. Depending on the tasks completion outcome the current states will be updated by the participants. The spectators get notified by the Helix agent if there are changes in the External view. A notified change will be read by the Spectators which will then perform their required duties. As a reminder and quick recap, a list of participants and spectators is maintained in ZooKeeper by Helix. Controller will be notified by ZooKeeper also if any node dies, so the controller can deal with it accordingly (Apache Helix, 2023; Gopalakrishna, et al., 2012).

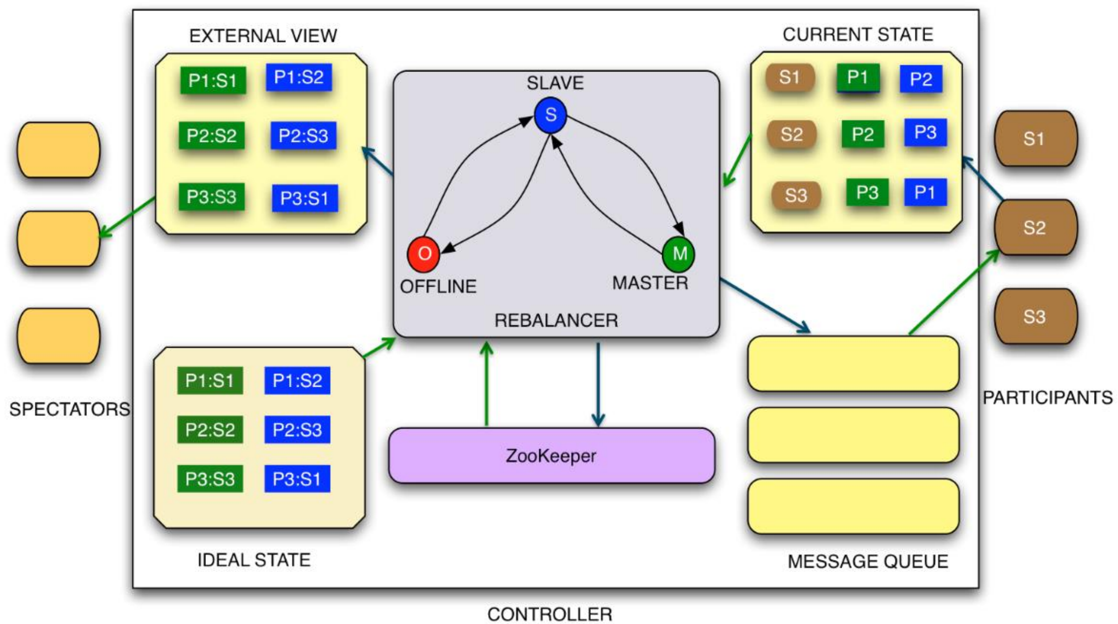



Figure 28. Interaction between Controller, Participant and Spectator roles (Apache Helix, 2023).

### 5.3 Kubernetes (K8s/Kube)

Kubernetes is a well-known open-source platform for orchestrating containerized application deployment, including automated scheduling and scaling the application when necessary. Kubernetes was originally developed by Google and it was open-sourced in 2014 (IBM, 2023). Kubernetes in its essence is supposed to give developers more agility, velocity and efficiency. These ideas I have explained already in Section 3.4 Containers and containerization, but let me give you a brief recap in the context of Kubernetes.

The idea of adopting Kubernetes and containers is to encourage us to build distributed systems with immutable infrastructure. The principle in the immutable system in this context is to not modify the artefact, container image, once it's created. A brand new image is created instead of having incremental updates for the old one. With a single operation, the old image will be replaced with the newly built one that contains updates and fixes for the application. It is easy to rollback to the old image if needed as that one is still in the registry. There's also a better record of how the new image was built so debugging is also easier (Hightower, et al., 2017). Just like Apache Helix, Kubernetes also uses declarative configuration to define the state of the application, and similarly, with Helix, Kubernetes also monitors these states and takes corrective actions if something goes wrong.

Kubernetes itself is a very big topic, and an entirely separate thesis could be probably made just on all of its functions, services etc. Obviously, going through the whole tool

won’t make sense and is not necessary, hence I have picked certain main topics of it that give some basic understanding of the tool related to this thesis’ content.

### 5.3.1 Basics

To understand Kubernetes architecture, we have to first know some basic Kubernetes-related glossary, the following Table 3 has definitions from Kubernetes’ own documentation.

| Term | Definition |
| --- | --- |
| **Container** | Image that usually runs an application or service. Typically, a Docker image. |
| **Control plane** | “Container orchestration layer that exposes the API and interface to define, deploy, and manage the lifecycle of containers”. Dubbed as the brain of the Kubernetes cluster. Historically known as master/head node. |
| **Node** | Worker machine that has local daemons, processes that run in the background. |
| **Pod** | Smallest Kubernetes object and also the simplest. A set of running containers encapsulated by Pod. |

Table 3 Basic Kubernetes glossary (Kubernetes A, 2023).

The Kubernetes workflow in a simple form, illustrated in below Figure 29 starts with client interacting with the Control plane’s API server component with an HTTP request. The Control plane delegates the work to worker nodes that carry out the tasks. Client is never interacting directly with the worker node.

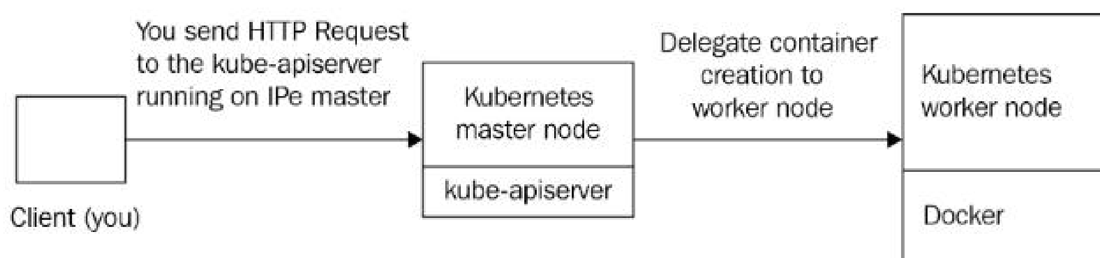



Figure 29. Basic Kubernetes workflow (Kebbani, et al., 2022).

The client is using Kubectl, Kubernetes' own command line tool, for sending the requests. Instructions sent to the control plane can be written in imperative syntax, which are basically shell commands. The other way to do it is through declarative syntax by either writing JSON or YAML files. YAML language is more popular among Kubernetes usages because it uses simple "key:value" syntax. While imperative syntax is rather easy, and some operations can only be done by it, declarative YAML files suit better for any bigger and repetitive operations. The YAML files can be version controlled and it also supports multiple resource declaration (Kebbani, et al., 2022). A simplified architecture view of Kubernetes components and their interaction with each other can be seen illustrated in below Figure 30. The following Subsections will explain the parts in more detail.

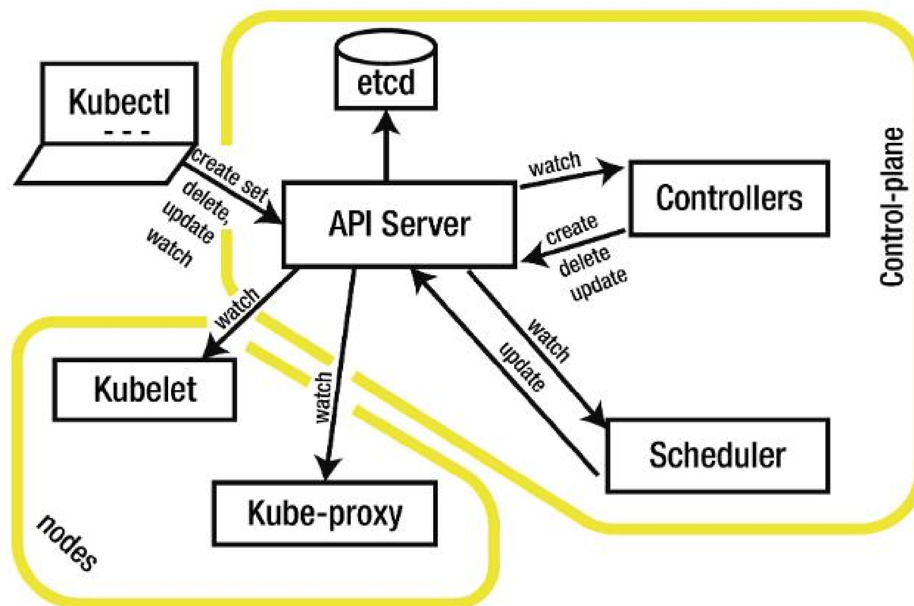



Figure 30. Simplified Kubernetes architecture view (Philippe, 2023).

### 5.3.2 Pods and Pod scheduling preference

The simple definition of pod was given already in the previous Subsection's Kubernetes glossary. Pods represent the basic building blocks you can have in Kubernetes and they're the smallest deployable artifacts in the Kubernetes cluster. Basically, when you have a pod, the containers in it always belong to one same worker node/machine. Containers are designed to only run one process, this keeps them easily manageable. Pod is a higher construct that allows tying containers together so they can be managed as one single unit. With pods, we can avoid putting multiple processes in one container to manage the group easier (Lukša, 2018).

Pods can have *topological spread constraints* configured, which allow controlling how the pods are spread across e.g., regions, zones and nodes, also known as failure-domains

or some other topology type defined by user (Kubernetes B, 2023). On top of topology spread constraints, one can set *node affinity* and anti-affinity properties to pods, which would make them be attracted or repelled by certain Kube nodes. These affinity and anti-affinity properties can be set as preference or hard requirement. Pods can have *tolerations* specified to them, where user would specify a key, value and effect. The matching specification will be added as *taint* for a node. With taints and tolerations, we can ensure that Kube scheduler won't be scheduling pods to unwanted nodes (Kubernetes C, 2023).

### 5.3.3 Control plane components

We mentioned the *API server* component already in the basics chapter, which exposes the Kubernetes API. The API server, also known as kube-apiserver, is stateless and relies on another component, *etcd* – a database engine, to store the states of the resources. Kube-apiserver can be scaled out freely thanks to it being stateless. The datastore etcd is NOSQL distributed database that stores the cluster state information such as what docker image is used, the pods' names, which machine the containers belong to and how many containers have been created. This etcd datastore is its own project not developed by Kubernetes, but something that is needed in Kubernetes setup in order for it to work. Essentially, whenever a client is calling read or write operations, the request is proxied through the kube-apiserver to or from etcd (Kebbani, et al., 2022). The below figure is a "zoomed-in" illustration of Figure 29's Kubernetes master node.

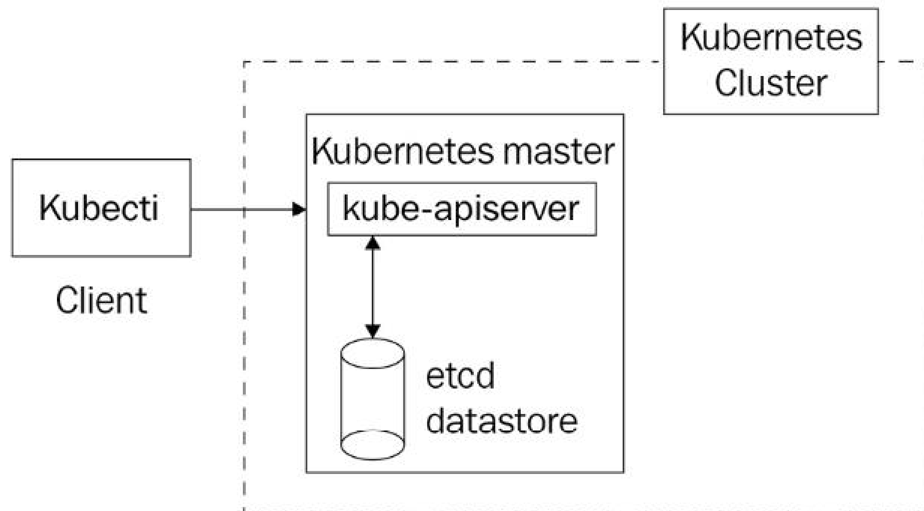



Figure 31. Kube-apiserver acts as a proxy between the client and the etcd datastore (Kebbani, et al., 2022).

Control plane also contains a Kube scheduler, Kube controller manager and cloud controller manager. The Kube *scheduler* selects a node for newly created pods without nodes. Controller processes are run in the Kube controller manager. *Controller* regulates the state of a system by tracking at least one type of Kubernetes resource. The controllers

make the resource's current state come closer to its ideal state; the controller manages the *control loop*, regulating the state of the system continuously. *Cloud controller manager* lets one link a specific cloud provider's API to one's cluster. The components that interact with the cloud platform are separated by the cloud controller manager from components only interacting with the cluster (Kubernetes A, 2023).

### 5.3.4 Node components

The worker nodes have components that run on every node. These components provide Kubernetes runtime environment and maintain the running pods, i.e., these components take care that the instructions received from the client are executed accordingly and end up in containers running in worker nodes (Kubernetes, 2023; Kebbani, et al., 2022). *Kubelet* makes sure containers are run in a pod and is considered a daemon itself. Kubelet is the one that interacts with the worker node's local container engine daemon like Docker. It is good to keep in mind that Kubelet is only able to manage containers Kubernetes has created, so if one bypasses Kubernetes and manually creates containers on the worker nodes, Kubelet cannot manage these. This is due to the fact that manual actions won't be reflected on the PodSpecs configuration stored in the etcd datastore, which is essentially provided to Kubelet. Another component run on a worker node is the *kube-proxy*. The proxy component partly makes the service discovery possible, so clients are able to interact with a set of pods made available on the network (Kebbani, et al., 2022). This concept is part of the Kubernetes Service method which I will explain in the following subsection. Kube-proxy maintains the nodes' network rules.

Now that we have gone through the Kubernetes basics and explained the important components, it is much easier to understand the Kubernetes architecture depicted in below Figure 32. One can see that the API server has a crucial role even though it's essentially just a REST API. The figure is showing a single master and worker cluster, by adding more nodes in both groups we can achieve high availability.

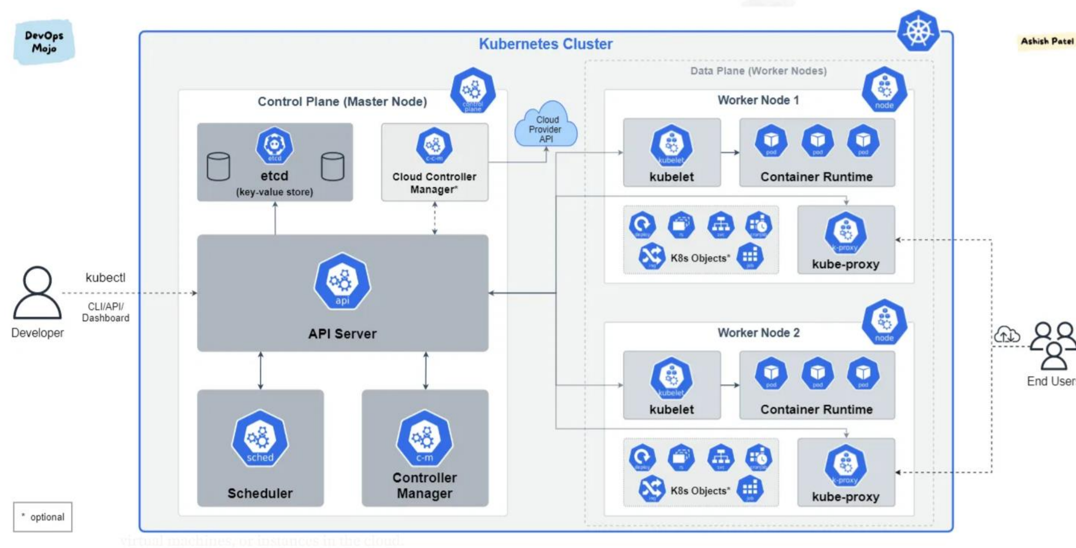



Figure 32. Kubernetes cluster architecture (Patel, 2021). View Appendix A for a larger version.

### 5.3.5 Service

Kubernetes has its own method for making network application's service discoverable and it is called Service in Kubernetes. From one of the earlier chapters, we learned a little bit about Pod networking and IP addresses and how easy the cluster inner communication is, it is generally advised to avoid relying on IP addresses for accessing a pod directly. The pods are ephemeral meaning that the IP addresses essentially change whenever a pod is recreated. The solution is to use Kubernetes Service resource, which as most of the Kube objects, can be deployed by using shell commands or declarative files (Kebbani, et al., 2022). The Service component essentially makes it possible to have a static IP address, and this happens by putting a Service in front of each pod to act as a traffic-serving proxy. The Service gets a static DNS during its creation which won't change as long as the Service is in the cluster. The Service creator's job is to tell it which pods the said Service is supposed to serve traffic to, and this can happen for example with selectors and labels. Service will select pods according to the labels assigned to them (Patel, 2021; Kebbani, et al., 2022)

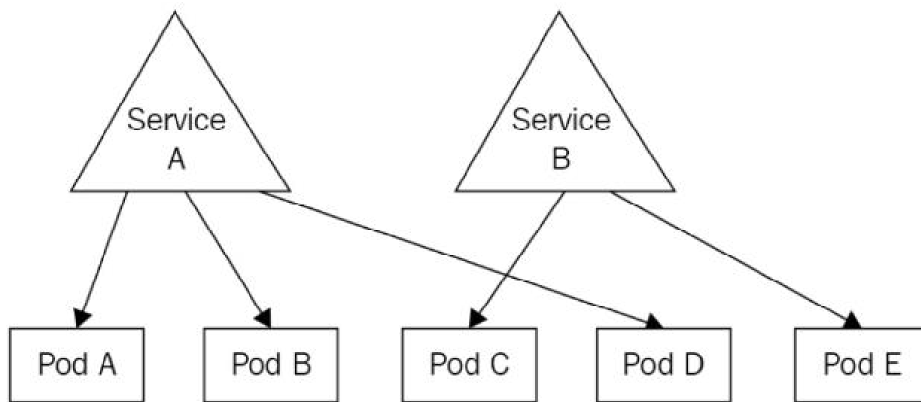



Figure 33. A simple depiction of Services exposing pods (Kebbani, et al., 2022).

Kubernetes uses a round-robin algorithm for load balancing, which means that requests are load balanced evenly to the number of pods that are behind the Service. E.g., From the above Figure 33 if we think of Service A as a load balancer with the three pods (A, B, D) behind it, each of the pods will receive 33% of the Service's requests. Service B would divide it 50/50 among the two pods (C, E) behind it. There are different types of Services, which I won't however elaborate more on in this thesis.

### 5.3.6 StatefulSet

One way to divide applications is to categorize them into stateful and stateless applications. Stateless applications don't store past knowledge of old transactions, it doesn't need to maintain the state and losing it is acceptable. Transactions in stateless applications can be thought of as vending machines, there's one request and a single response to it. When in need to save state and refer to past transactions, we have Stateful applications. Stateful applications save the history and context, past transactions affect the current transactions, e.g., online banking (Red Hat, 2020).

Kubernetes StatefulSet is a Kube native way to manage stateful applications. It is a workload resource that helps manage the deployment and scaling of a set of Pods. The pods of StatefulSet have a sticky identity, meaning that while they're created from the same specifications, each pod has its own persistent identifier that is kept across rescheduling. Basically, the pods are guaranteed to be unique and ordered. The pods have an ordinal index: when we have N number of replicas in StatefulSet, the pods will be assigned integer ordinal starting from 0 and ending with the final pod being N-1. When the pods are being deleted, the process starts in reverse order so from {N-1..0} (Kubernetes D, 2023).

# 6 Product Architecture

This thesis is a commission to Relex Solutions company where I have been working as DevOps engineer before and during my master studies. We will hereafter refer the company as Relex. Relex focuses on developing software solutions for supply chain and retail planning management. They have a product called Plan which is a distributed system with a monolithic architecture. The product has the capability to run multiple monolithic instances simultaneously for a single application. In practice, this means running multiple Plan instances with slightly differing configurations, or 'roles', in a single clusterized application, this is how scaling out is made possible. Relex is planning to start to use Kubernetes as an orchestration platform for this Plan product. In this chapter, I will describe the current architecture and explain the motivation for Plan to move to use Kubernetes.

While this thesis focuses more on cloud computing, the Plan product itself is something that started with supercomputing qualities. Due to the requirement to keep up with modern times, needs and solutions, the project is slowly adopting cloud computing technologies and frameworks.

The Plan product is a SaaS solution that is run in a private cloud. It is a single-process JVM monolith, illustrated in Figure 34, that has been built mostly with Ruby, Java, and Scala. It has the user interface layer, business logic and data access layers. Plan is considered as data-intensive application as it processes a lot of data. One thing to highlight about the product is that it also has an in-memory database as seen in the below figure, hence all the processing is done in-memory. The application itself is surrounded by multiple supporting services.

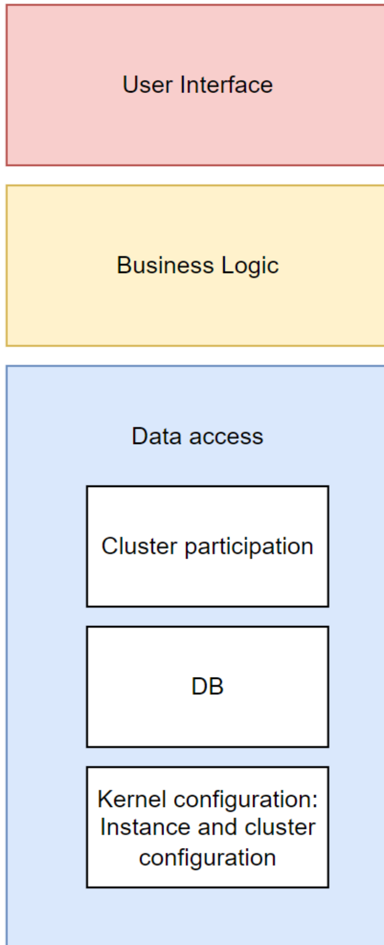



Figure 34. Relex Plan's monolithic architecture.

The product has kept its monolithic form and in-memory database over the years and will keep it so for a non-foreseeable period. There are a lot of dependencies in and between the data, splitting it wouldn't be conventional. Relex's solution allows in-memory calculation which vastly improves the performance and shortens the query time. In its simplicity: "data is queried when it's within the computer's random access memory (RAM), as opposed to being read on and off physical disks", explains Falck (2013), co-founder of Relex, in his article of "Big Data – Big Talk or Big results?". If we'd put supercomputing and cloud computing on the opposite sides of a spectrum of large-scale computing systems, Relex Plan's characteristics are actually closer to supercomputing, as illustrated in below Figure 35.

Figure 35. Relex Supply Chain Management (SCM) in the spectrum of large-scale computing systems, where supercomputing and cloud computing are the opposite ends.

## 6.1 Plan Roles and Helix

The Plan monolith has roles, see Table 4, that can be configured to cluster settings when scaling out. These roles allow for running multiple monolithic instances simultaneously for a single application. Helix coordinates which of the potential roles are active for which instance, i.e., if multiple instances have a role that can be active in only one instance at a time. Helix decides which instance acts as the active role, making all the rest instances with the same role go into a passive mode (for that particular role). The passive instances won't perform the functionality of that required role, but in case something happens for the active node, they're ready to take the responsibility and become "active" or the leader. This way the high availability is also implemented in the Relex Plan.

| Role | Description |
|---|---|
| Transactor | Writes commits to the distributed object storage. A single Plan instance is active with this role at a time. |
| Worker | Defines a worker instance that handle jobs. There can be multiple Worker instances. |
| Backend | Serves the user requests; acts as a backend that receives traffic from routing components. Many backend instances can be active at the same time. |
| Scheduler | Schedules and partitions jobs to worker instances. Only one active. Combined with Transactor in declarative-roles mode. |
| Authenticator | Provides internal authentication. Only one active. |

Table 4. Relex Plan cluster instance roles.

For Helix, the resources that its managing in this product are the roles, users that log in to the system and scheduled jobs. Each of these resources have their own state that are determined in the resource's state model. E.g., State model of 2 states for role X could be stopped and running. Remembering that Helix itself has types: participant, spectator and controller. The Plan's roles are participants in the Helix cluster, each configured role having its own participant which belongs to its own Helix cluster, e.g., Plan with one instance and three roles: transactor, worker and backend would have three Helix clusters, one for each role. Helix controller manages the states of the participants.

In Plan, the Helix does not split the resources into multiple partitions, which removes the trouble of needing to distinguish the difference between partition and resource. Plan always has one partition of each resource. Even though partitioning doesn't exactly happen, replication is still used. As described in the role table, Plan can have multiple worker and backend roles, when scaling out the resource is replicated.

It is important to note that there are basically two different clusters and not to get them mixed up: The Helix cluster defined in Helix, hereafter referred to as either as "Helix cluster" or "sub-cluster" and Plan cluster, which is basically one environment, with one or more instances. An example depicted in below Figure 36, shows a situation where we have one Plan cluster (one environment), with two instances. The two instances have slightly different roles, hence they also have a different amount of sub-clusters (Helix participants). The figure also shows how there's only one Transactor activated even though two instances has that role, which is due to the constraint that there can be only one active transactor. This leaves the Plan-instance-2's Transactor in standby mode, ready to take over in case Plan-instance-1 fails for any reason.

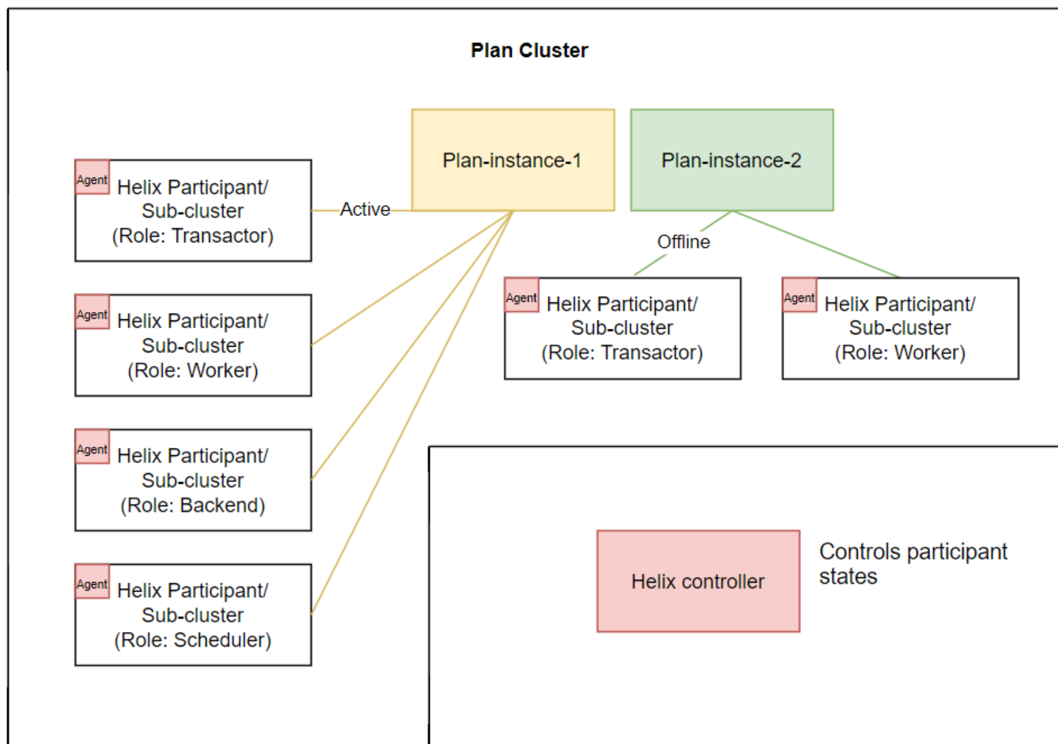



Figure 36. One environment with two instances, each of them having their own roles.

There's two ways (modes) to configure the Plan application role distribution, one way is *role-based* and another way is to define *declarative-roles*. For role-based configuration, the roles are manually assigned for each instance before they are created. In declarative-roles we declare how many instances with certain roles we want to have, e.g. Two instances with worker-role, and Helix manages the assignment. For this thesis, it is enough to know the very basics of roles and the mentioned difference between the two role modes.

## 6.2 Distributed commits – component communication

Together with local in-memory database, Plan also has distributed database – object storage. Figure 37 shows the simplest form of transaction in a Plan, with only one active instance. Transactor writes a new snapshot and sends it to object storage and will then proceed to update the timestamp, to ZooKeeper as well. The timestamp in this case is the time of each transaction, a unique ID that allows identification of the elapsed time between two requests. The timestamps will be used for comparing the order of the transactions, as in which is older and newer. Thus, when a node detects that there are new changes available, it'll retrieve it from object storage.

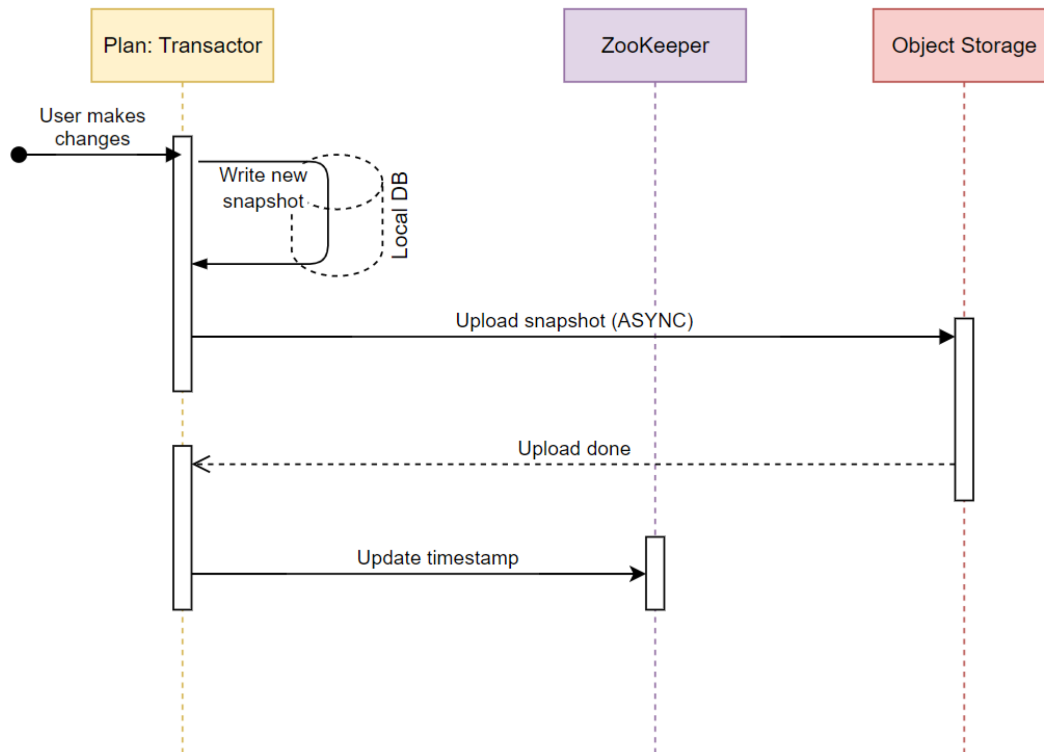



Figure 37. Plan single instance transaction.

For multiple instances to make changes to the distributed database the single instance with an active Transactor role writes all the changes to the database, other instances send changes to that Transactor over gRPC protocol. Instances with Worker and Backend roles send change requests to the Transactor instance which rebases (if needed) the changes on top of the current database state and commits them into the database. Transactor writes a snapshot to the local server disk and then proceeds to upload it to the object storage. When the upload operation is complete, the cluster state in the ZooKeeper cluster is updated to reflect this change. Other instances in the same cluster observe this change in the ZooKeeper state and download the required snapshots from object storage after ZooKeeper notifies that there's a new change. Basically, timestamp comparison is done, and if the ZooKeeper timestamp is newer than the current, the instance will get the new changes from object storage. The more complex transaction logic is depicted in below Figure 38 with two instances.

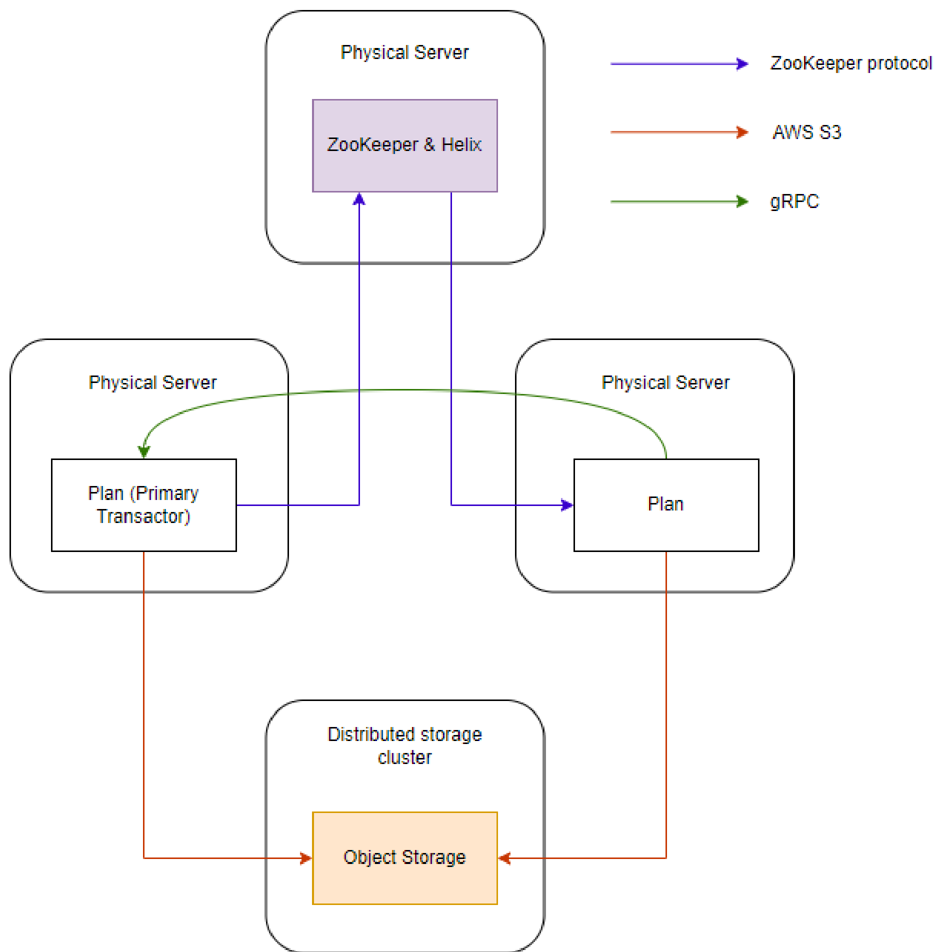



Figure 38. High level picture of transaction in Plan, depicted here with two instances located in different servers.

To refer back to subsection 4.1.1 Leader and Follower, the transactor model used in Plan is basically leader-follower type of replication, where the active Transactor-role represents the leader, who solely is allowed to make changes to the system. All the other instances are followers, who receive the information of new changes and can ask them from the leader. For a more detailed sequence-based diagram of the multi-node transaction depicted earlier, please look at Figure 39.

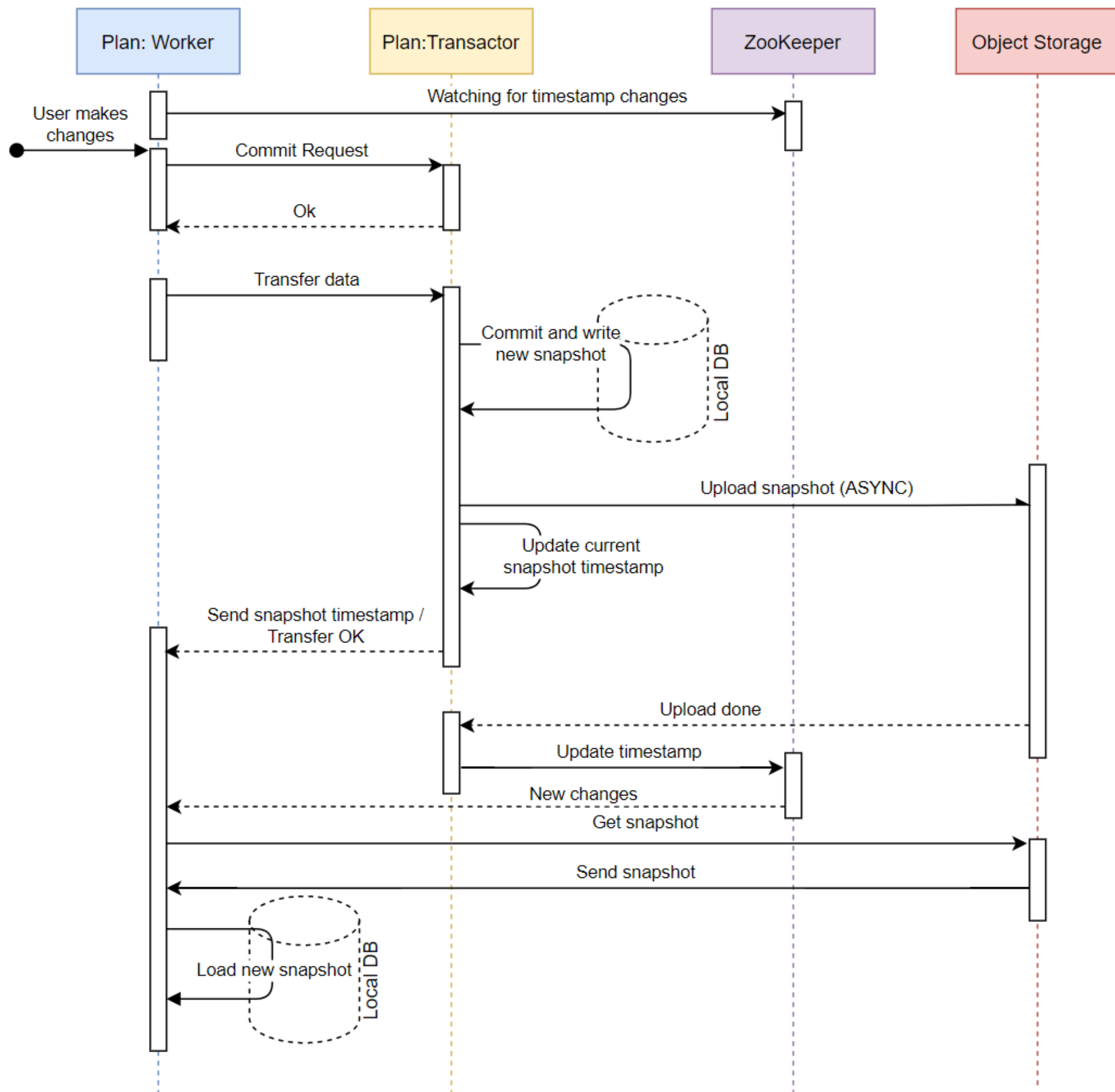



Figure 39. Sequence diagram of transaction in Plan with two instances. Worker role transfers user made changes to Transactor, which writes it to the local database and uploads it asynchronously to object storage. Worker node will receive information of new available snapshot, fetches it from object storage and loads it to the local database.

## 6.3 Updates and migration

The current architecture requires Plan to have some downtime so that data migration can be done to the database. In practice what happens is that the Helix role called MaintenanceParticipant joins the cluster, and tells all the Plan roles to go to OFFLINE in order to run the data migration tool. Data migrations need to be done in case some database structure, schema changes, has happened. With the current architecture, the application will be stopped to run the migrations. The downtime is necessary for the migration run because otherwise, users would experience errors if database updates were done for live

instances. The root cause of the errors is that the current updates are not backwards compatible and hence with the current update and migration model, we can't really have no-downtime updates.

### 6.4 Motivation for Kubernetes orchestration

Relex Plan production environments run on bare metal servers and operators do capacity planning manually, which requires a lot of effort and is error-prone. Having an orchestrator in place would provide several benefits, such as automatic scheduling workloads onto a shared pool of resources. Better isolation between customers. Giving development teams end-to-end ownership of their service.

Many things are currently custom-built or done manually, which the Kubernetes platform would essentially bring relief to by reducing the amount of work needed to manage the growing number of product environments. The common pool of resources is a big driver to use Kubernetes. When there's a common pool of resources, Kubernetes' scheduling and eviction concepts are very useful to categorize node types and constrict which nodes accept what kind of pods and the amount of them. These can be used to differentiate server types, e.g., we can use certain hardware types only for large customers that may need entire nodes' worth of capacity. Topology spread constraints can be used to further control where specific workloads are scheduled within multiple regions and zones. For example, this could be used to schedule a Plan workload in the correct data center where its ZooKeeper instance is located. Overall, this resource pool concept benefits end users through better reliability and fewer outages and maintenance breaks.

The long-term goal seems to be to essentially build some type of PaaS solution that'll be used for setting up and orchestrating product environments. The below Figure 40 illustrates what the Kubernetes architecture looks like with a multi-node setup.

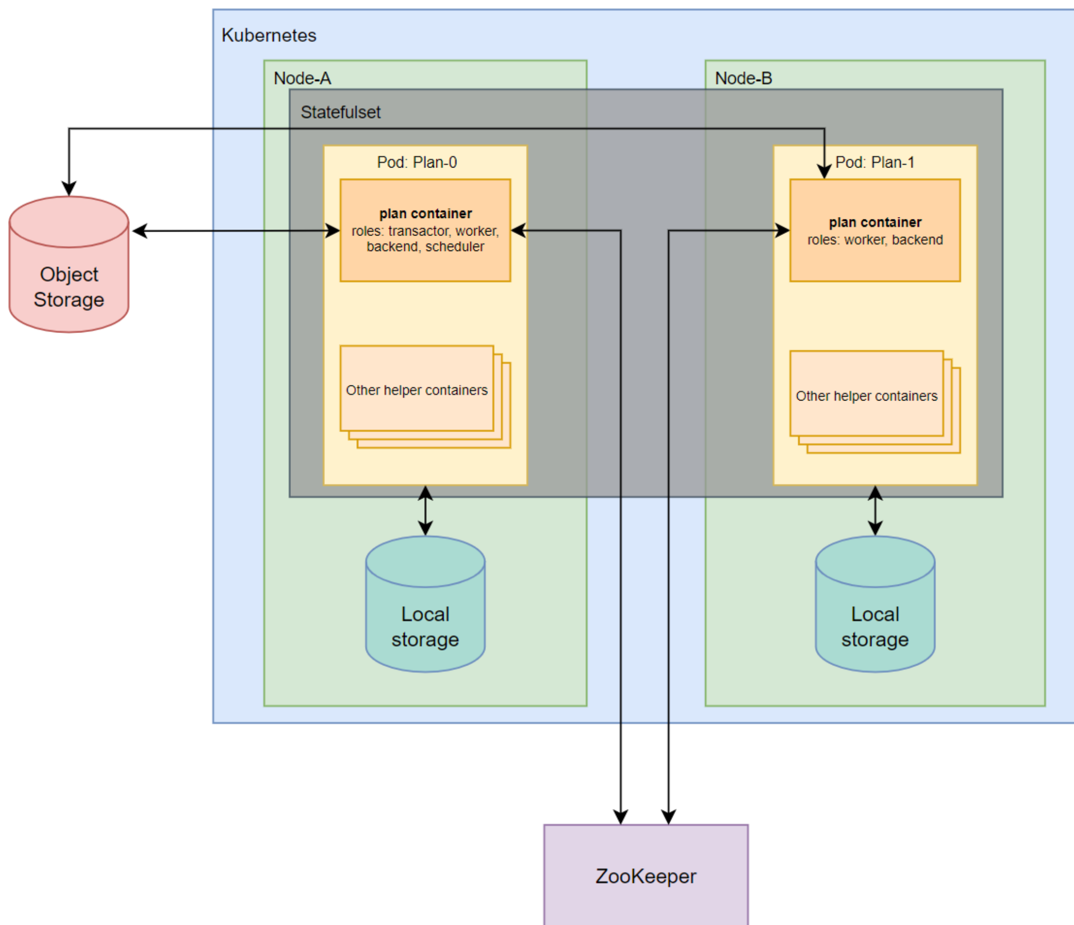



Figure 40. Kubernetes architecture with Plan of two instances, which reside on different servers (nodes). The roles listed in the plan containers are just example of how the distribution might go in multi-node setup.

# 7 Research questions and methodology

This thesis's main theme or topic is to ensure smooth operations with a monolith Plan using Kubernetes with the existing Helix cluster management. This is scoped into two main research questions:

1. How would Plan environment's lifecycle management work with the Kubernetes orchestrator?
2. How to handle Plan cluster orchestration with the Kubernetes platform?

The questions are more on a higher level as the product's enterprise architecture is too big to handle entirely. We will tear down these two questions into more manageable pieces - scenarios, that the client is interested in finding solutions to as part of this thesis. How does Plan work in Kubernetes "world" when we want to:

1. Add a new instance to an existing cluster
2. Stop an instance
3. Restore instance
4. Empty a server
5. Update Plan while taking into account database schema changes
6. Rollback an update

The main methodologies used in this research are literature review and design science, for the latter I'll be adhering more loosely and mainly use it as guidance for the research process. As Kubernetes is a well-documented tool, existing literature regarding it will be used, including but not limited to the official Kube documentation and other book publications. The concrete outcome of this work are several solution design diagrams with documented description of them, these are evaluated and approved by Relex Plan people that are overseeing this work and acting as the product experts and instructors. These outcome characteristics and the needed process for it are similar to what is defined in design science methodology (Hevner, et al., 2004).

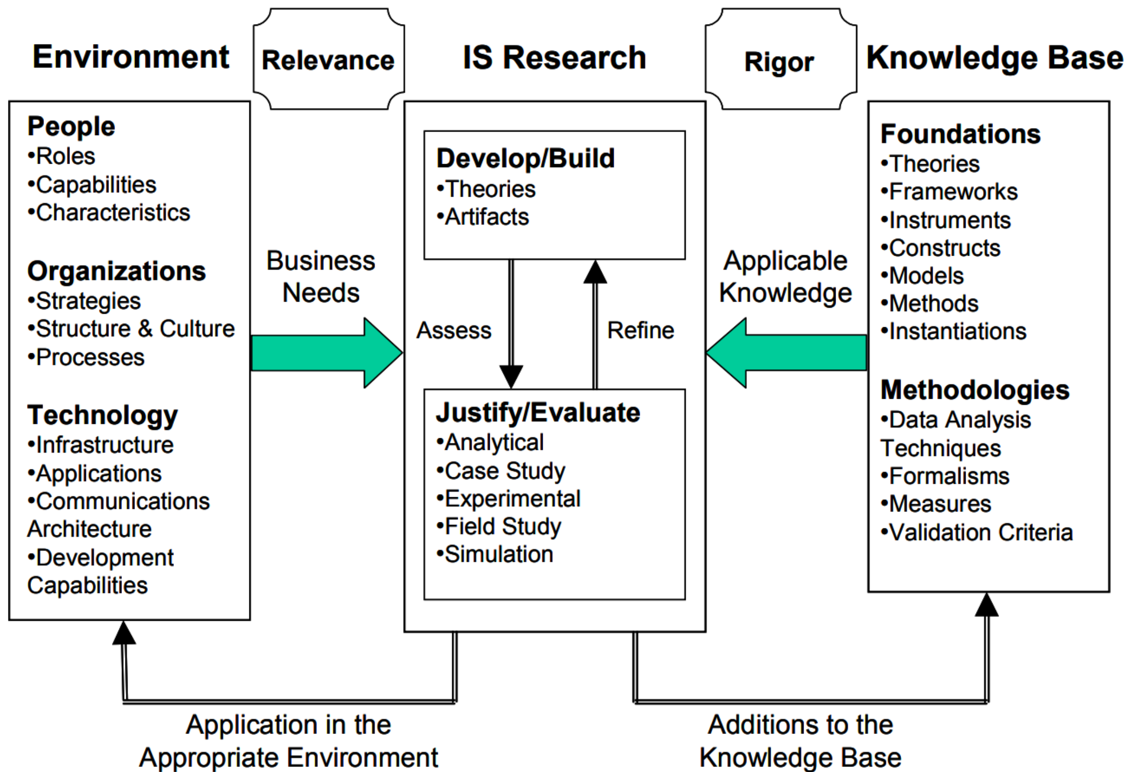



Figure 41. Information Systems (IS) Research framework used in design science methodology (Hevner, et al., 2004).

The design science defines seven research guidelines that applies in my research, see the following Table 5 for application details. The above Figure 41 shows the high-level framework that is used in the design science, where in this thesis' case the commissioner, Relex and the product Plan represent the left side "Environment" of the figure. The right side of the figure are basically the literature review sources that I am using, but also the existing foundations and methods that Relex Plan has, and what the "Technology" is built on, which I have introduced in Chapters 1-7. The middle part of the Figure 41 is represented in this thesis by the actual research work.

| Guideline | Description | Application in this work |
|---|---|---|
| Guideline 1: Design as an Artifact | Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation. | Solution ideas, documented in written format and explanation supported by diagrams |
| Guideline 2: Problem Relevance | The objective of design-science research is to develop technology-based solutions to important and relevant business problems. | Smooth operations using Kubernetes. How to handle Plan cluster orchertration and manage applications's lifecycle |
| Guideline 3: Design Evaluation | The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. | Analytical & Descriptive methods: Regular meetings with Plan experts to evaluate provided solutions, corrective actions will be done from given feedback |
| Guideline 4: Research Contributions | Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies. | Main contribution as design artifact (see Guideline 1). Foundation contribution might come as sideproduct during discussions. |
| Guideline 5: Research Rigor | Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact. | Literature review method used |
| Guideline 6: Design as a Search Process | The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment. | Basically Generate/Test cycle used. Short iterations of finding and designing solutions and "testing" is by getting them evaluated by experts. |
| Guideline 7: Communication of Research | Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences. | Approved by comissioner |

Table 5. Design Science Methodology guidelines (Hevner, et al., 2004), and how they are applied in this thesis work.

This thesis will be done by working closely with the Relex experts, consisting of architects and senior developers. Weekly meetings will be organized to discuss about the earlier mentioned scenarios, so the work is basically done in small one-week iterations. The meetings will act as platform of discussing the matter but also for evaluating and giving feedback on my suggested solutions.

# 8 Design suggestions

Originally, we had an idea to make a custom Kubernetes operator for Helix Admin API which would then be used via Kubernetes API to manage Plan orchestration. This idea was scratched eventually in favour of using Kubernetes' StatefulSet object, explained in Subsection 5.3.6. Figure 42 shows the original design ideation with Helix Admin as Operator. The reason why this idea was abandoned was that it would have needed more effort to make the operator while StatefulSet is basically a ready-made solution and native to Kubernetes, which is a big plus. We should use as much of Kubernetes' own solutions as possible when trying to adopt Kubernetes orchestration in the company's DevOps. The updated design, Figure 43, was already partly shown in Figure 40 in Subsection 6.4. But the following figure can be viewed for simplified high-level design.

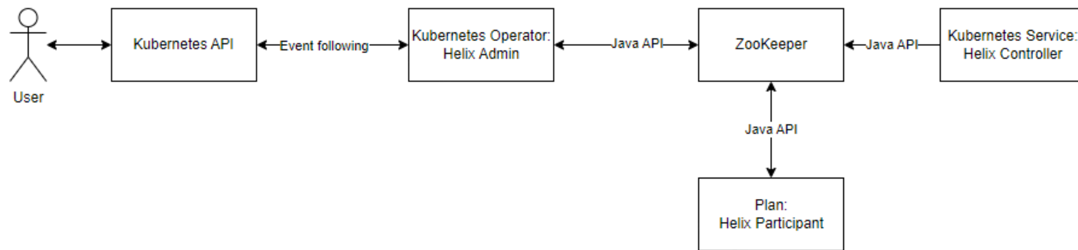



Figure 42. Original rough high-level design.

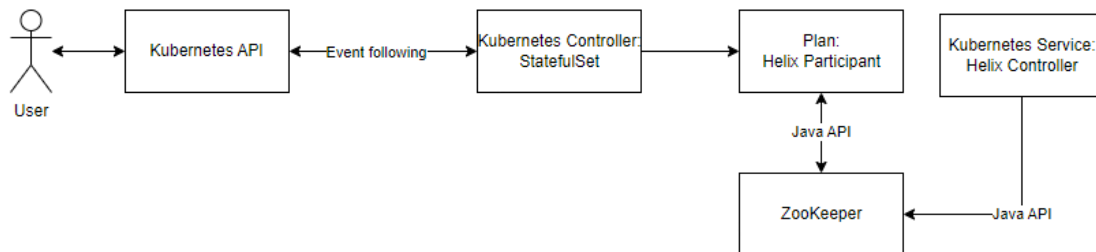



Figure 43. Updated high-level design.

## 8.1 Adding new instance – Scaling out

Setup: We have an existing Plan cluster running in Kubernetes, how do we add a new Plan instance?

This scenario is quite simple as there are no Helix operations needed. We can simply tell the StatefulSet to have more replicas, as depicted in the following sequence diagram, Figure 44. With certain node affinity and/or taint rules, it can be made sure that the scheduler knows which server the new instance is placed to.

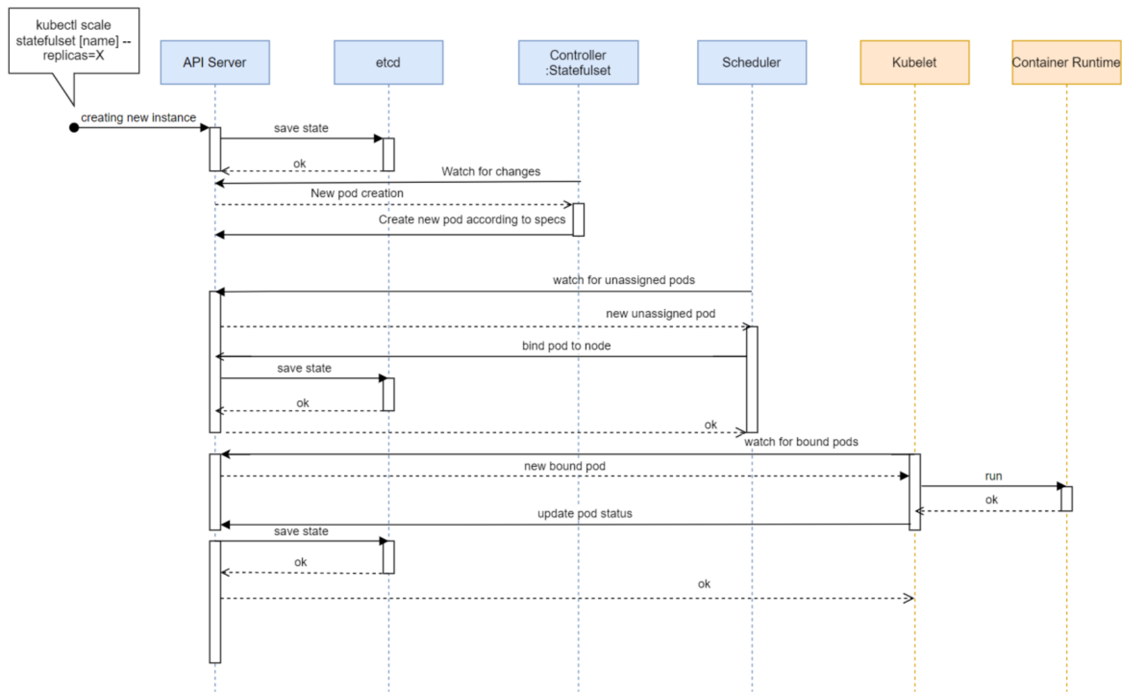



Figure 44. Scaling up the StatefulSet to have more replicas. Message reaches container runtime which does that actual setup.

## 8.2 Stopping an instance – Scaling in

Setup: We have an existing running cluster of Plan, minimum two instances. We want to scale in by 1 instance.

The target is to always do shutdowns gracefully, meaning that we won't abruptly close anything but wait till the instance is ready to be shut down. This kind of operation requires Helix operations. We want to make sure that Plan has time to prepare itself for shutdown, meaning that it will finish any ongoing work but at the same time also make sure that graceful shutdown is applied to Kubernetes components. There are basically two shut-down processes: Plan and Kubernetes.

The Plan side of shutdown has to be done with a pre-stop hook that Kubernetes provides. This is due to the fact that in Kubernetes' own shutdown process, the connections will be terminated, which are most likely still needed on Plan's side for its own graceful shut-down. According to the Plan experts, Plan's graceful shutdown process might take days if there's a lot of data to process, hence pre-stop hook and the custom grace period is a good combination. The custom grace period defines the time that is given to both the prestop hook and Kube's graceful shutdown in total. Regardless of if everything went

accordingly, the Container Runtime will be told to send a KILL command to stop everything at the end of grace period. Figure 45 below shows the timeline depiction.

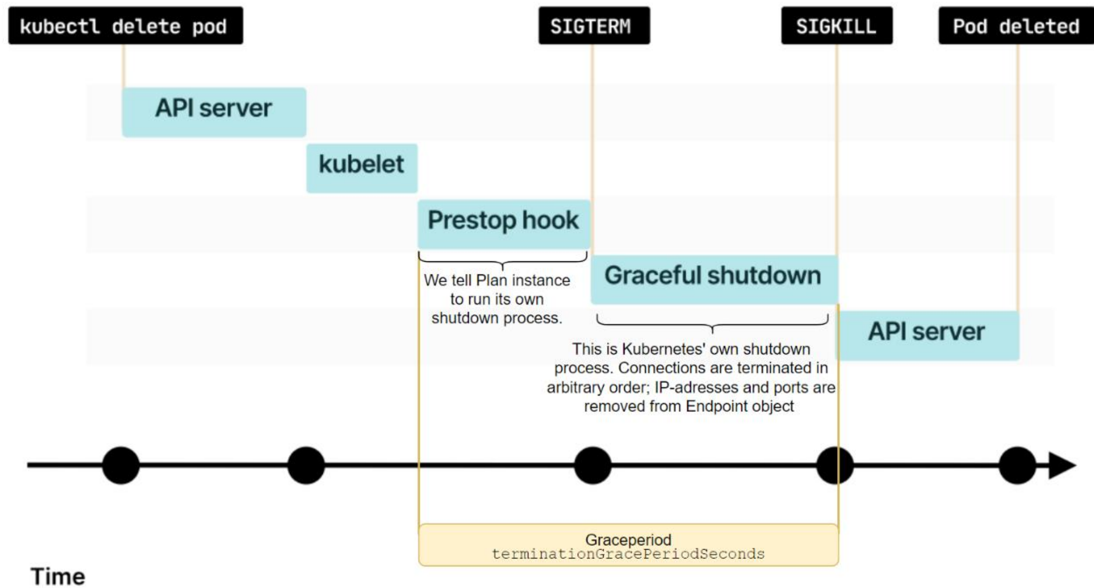



Figure 45. Graceful shutdown's timeline from Kubernetes' point of view, an adaption of (Polencic, 2020).

Plan is being informed that one of its instances will be shutdown, so it tells itself (or that particular instance) to start shutting down itself. In the Helix subcluster, Helix controller will be telling its participants to start moving to OFFLINE state. In Plan, certain roles have a "Lame duck" state that is between OFFLINE and ONLINE; this state basically acts as a buffer for ongoing tasks to finish while already denying any new users and jobs. The details of what happens during the pre-stop hook are depicted in below Figure 46.

In theory, grace-period could be set for months, as Kubernetes is aware of its hook's status through the handlers; when Plan finishes its shutdown Kube will start its own process of shutting down the pod. Though setting the grace-period for too long means that if pre-stop phase gets stuck, Kube will not send KILL command until the grace period ends, and manual intervention is needed. This has it's good and bad sides; abrupt shut downs might not ever be wanted and we want to actually handle situation manually case-by-case. Bad side is that some better monitoring might be wanted to set in place, so operation admins can react faster. Either way, the grace-period should be sensible number; if Plan's maximum worst case shut down time takes e.g., three days, maybe give couple days of buffer time on top of it and not set the timer for weeks or months. Essentially, the company has its service level agreements with customers, so there's no point for ridiculously long grace-periods.

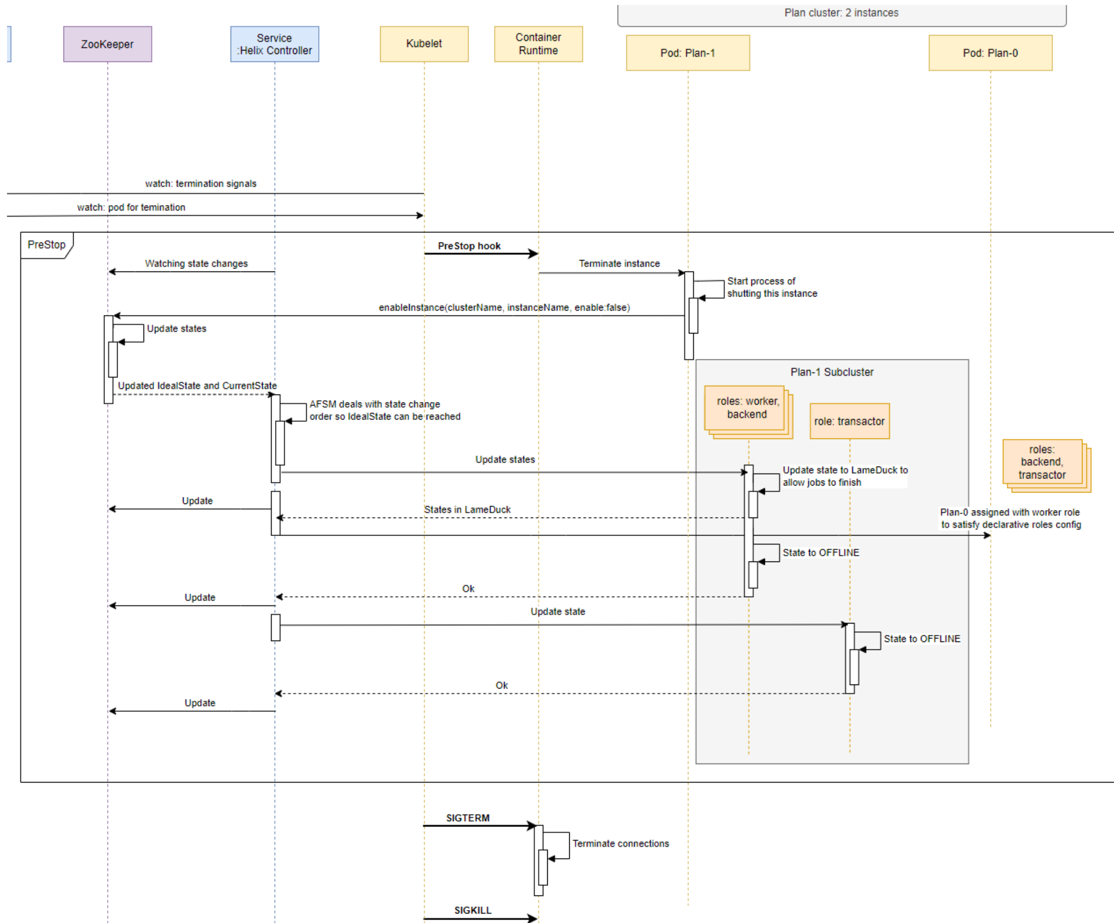



Figure 46. Partial sequence diagram of scaling Plan instances down by one. Note the Pod numbering, as mentioned in StatefulSets the stopping starts in reverse order {N-1…0}. In this example, Plan application is using declarative-roles. Refer to Appendix B for the whole diagram.

In some rare cases, more forceful stops are required, where we basically tell Kubernetes to kill the instance without any grace-period for the Plan itself, this is depicted in Figure 47. This scenario should always be avoided as it can lead to database corruption or other errors if there are ongoing processes, and the instance is just shut. It’s basically the same as pulling a plug.

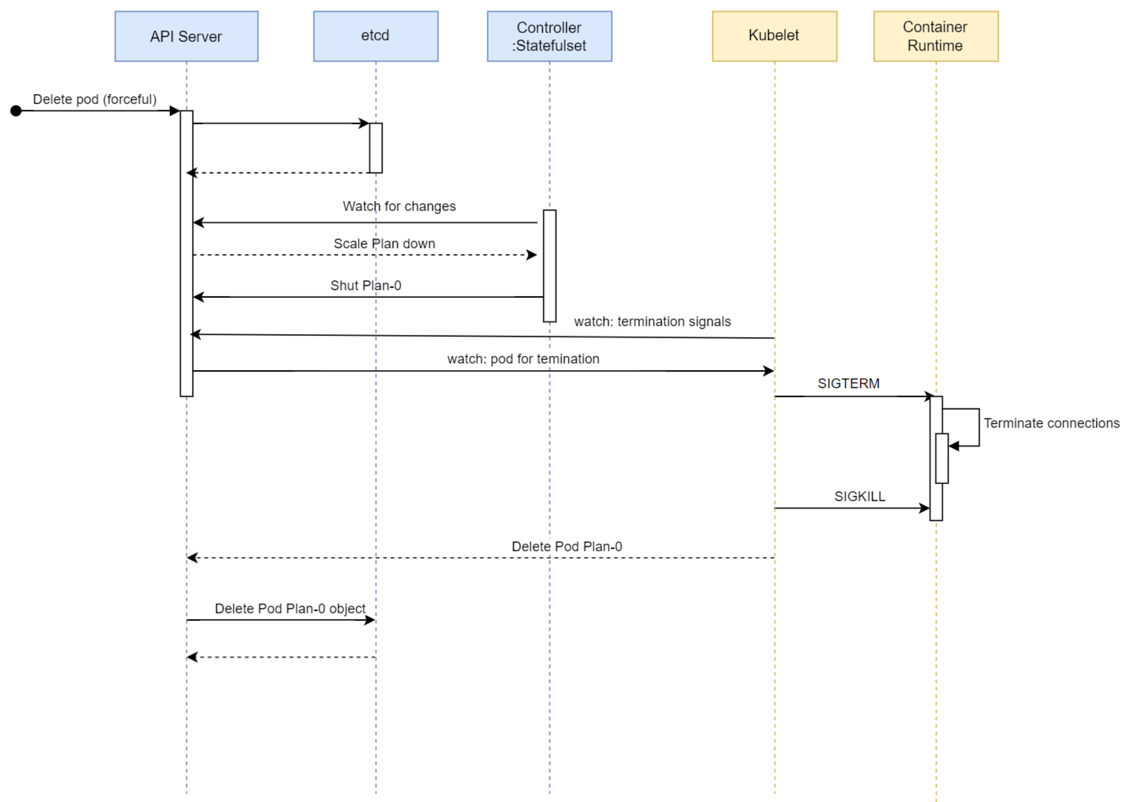



Figure 47. Forceful scale in as sequence diagram. Stopping instance by sending KILL-command without grace-period.

## 8.3 Restoring instance

The instance restoring we depict here is basically the high availability case. Let's say we have a case of server hardware failure, which takes down some Plan instance and makes it impossible for it to automatically recover in the now damaged server. What would happen in this case is that the Helix will get the information of Plan role imbalance and it will reassign the role that went down with the instance. The role placement is not random, but there's a rebalancing algorithm that calculates the best place, instance, for the role to be reassigned to. The StatefulSet controller will recreate the missing instance, which is possible because as it was mentioned earlier in Subsection 5.3.6, with StatefulSets the pods get a sticky persistent ID, so the controller knows exactly which pod went down and can restore or re-create the same pod. Kubernetes schedules the restored pod-instance to some new node from the resource pool that fulfils requirements or preferences (node-affinity, taints, topology constraints) that have been set. An example case is depicted in below Figure 48, where a hardware failure causes Plan-instance-3, or pod-3, to go down. The role earlier assigned to that instance gets reassigned to somewhere else until Plan-instance-3 gets recreated. To illustrate the sticky id, the Figure 48 doesn't have Plan-instance-2, Plan-instance-3 is restored as Plan-instance-3. In reality, for a regular instance

restoring, when Kube detects that there's a pod missing it would try to recreate a missing instance pod with correct ordinal number, in this case pod-2 or Plan-instance-2.

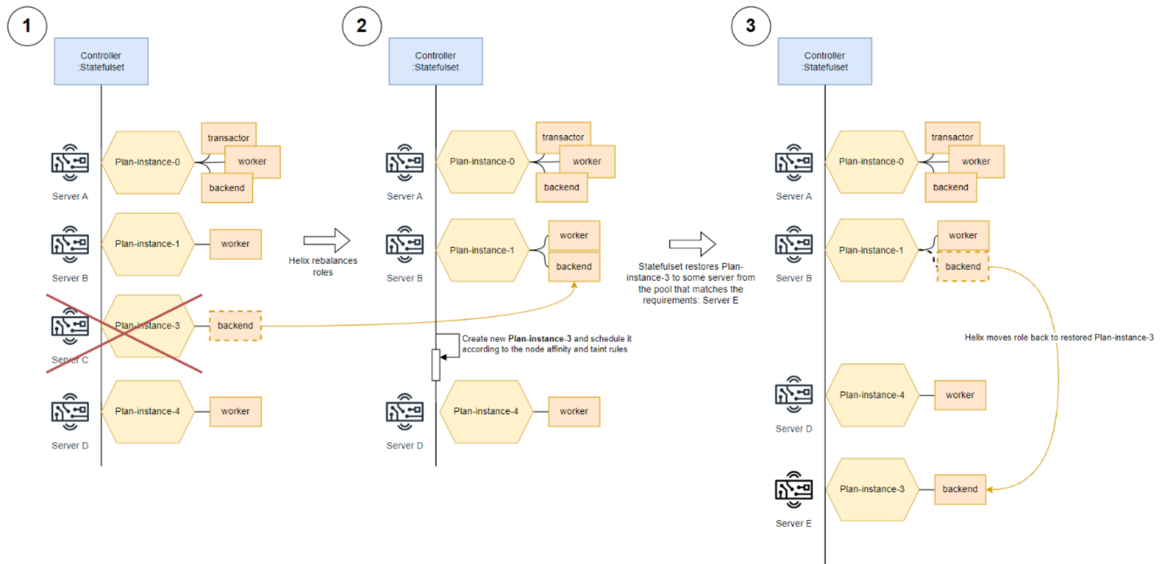



Figure 48. Instance restoring. Server C breaks, Helix rebalances the Plan role to some other instance, Plan-instance-1, in another server. Kubernetes' StatefulSet restores the Plan-instance-3 to another node-server picked from the pool, Server E. Helix notices that Plan-instance-3 is restored so it moves the backend role back to the instance.

For this kind of hardware failure depicted in above Figure 48, it is necessary to have a human operator to fix or replace the damaged server and eventually, when that happens, it will go back to the resource pool to be used. One thing that could happen, though probably it would be quite a rare case, is that for some reason the restored server is deemed more suitable for the Plan instance that got originally rescheduled for the new server. Thus, when the original server is restored, Kubernetes moves the instance back to it. This is depicted in the next Figure 49. A case why this might occur is that the installed "best-option" server is very customer specific so the resource pool itself has only so many of them. Hence, when one breaks, the instance is rescheduled to "second-best-option". When the server is restored, the top preference can be fulfilled again.

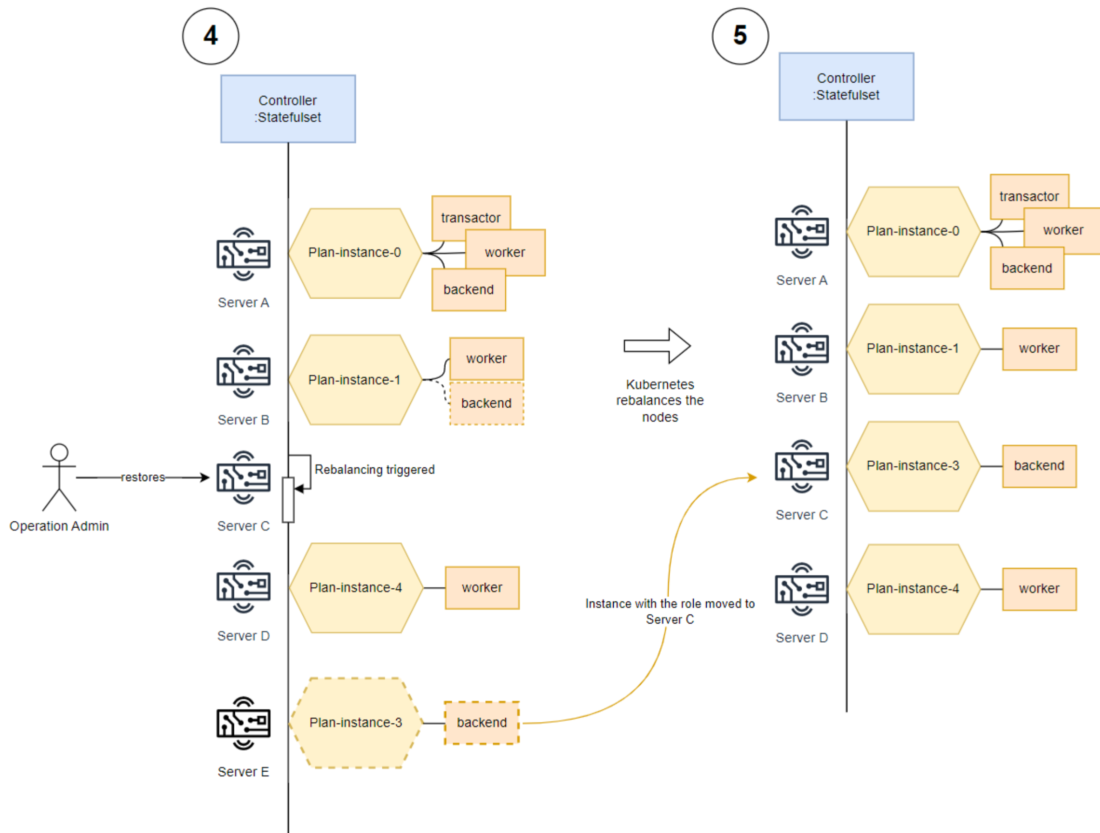



Figure 49. Server C restored from Figure 48 rare case. Operator fixes the broken Server C, and Kubernetes deems it more suitable for the Plan-instance-3 than the Server E that it was assigned to, so rebalancing happens, and the instance is moved back to its original server C.

### 8.4 Emptying server - Moving instances to another server

Emptying a server is about moving living instances to another server. This case has similarities with the previous use case when the whole server goes down and the instances were forced to be restored in another server. Instead of forcefully shutting the server and the instances along with it, we do the moving more nicely.

For Relex, availability is important, so it must be kept in mind that the moving should be done gradually and that there shouldn't be performance issues on the users' side when the moving is done for the live environments. This would mean that we'd essentially need to scale out in the target server first and then drain the source server. By initiating *node draining*, Kubernetes can be told to evict all the pods from the node. This is rather common to do for performing various maintenance tasks for the server, e.g., a kernel update or hardware maintenance.

Let's describe step by step what should happen when server emptying is done for maintenance purpose:

1. Figure 50, taint rule is defined for the to-be-maintained server/node, so when we scale out, that particular server won't be picked by the scheduler.

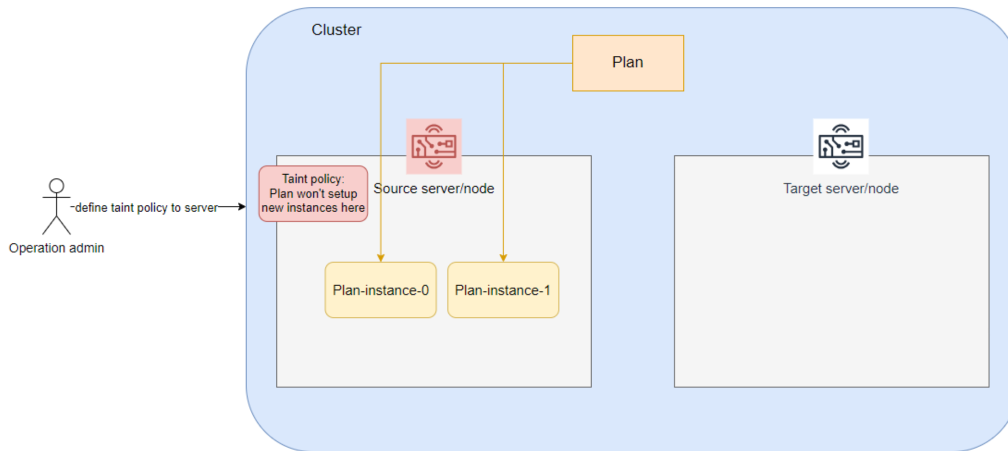



Figure 50. Taint rule defined to to-be-maintained server.

2. Figure 51, Plan is scaled out by the number of existing instances, e.g., if there are originally two instances, we scale out to four. This way we can assure the performance stays the same for the users. The new instances are scheduled to some new server from the source pool that fills the preferences.

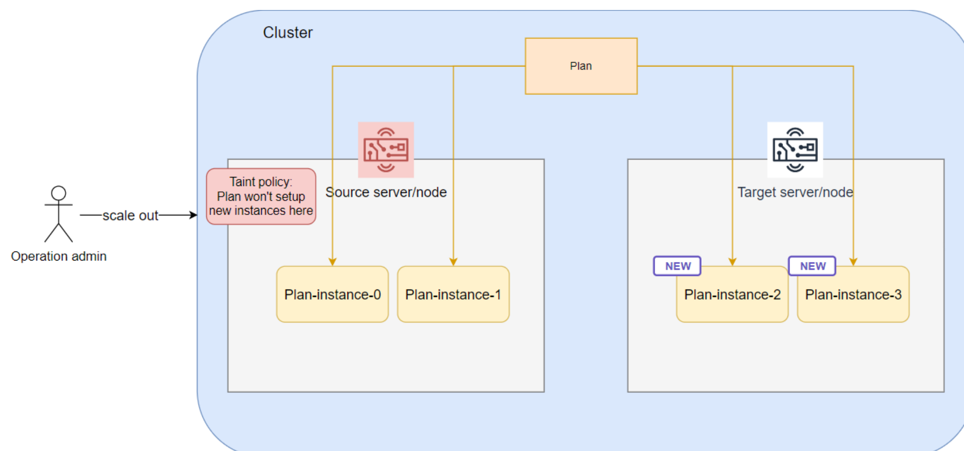



Figure 51. Scale out by number of existing instances.

3. Figure 52, the to-be-maintained server is drained. Drain command does the following: cordons the server so it is non-schedulable, deletes the pods in order by gracefully terminating them. Completion of draining basically decommissions the server, which will wait for human actor to initiate some maintenance.

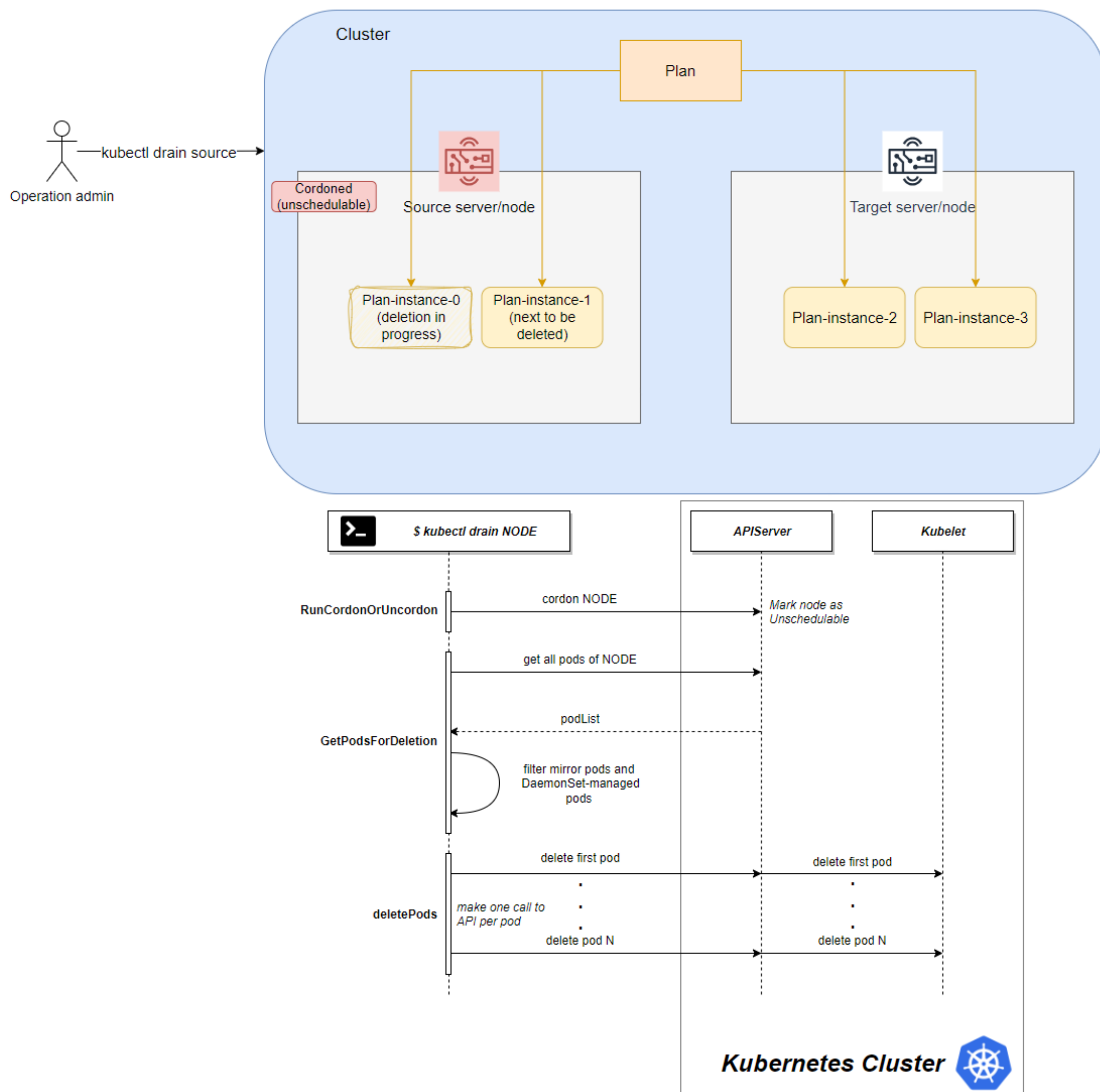



Figure 52. Drain to-be-maintained server. Sequence diagram from (Kubernetes E, 2023)

4. Figure 53, StatefulSet will try to recreate the pods deleted when we drained the server. As the old server is on the taint list (and also non-schedulable), the pods will be scheduled to some other server. As StatefulSet pods have the sticky id, the pods with exact same id are recreated, i.e., StatefulSet knows which pods are missing and it knows to recreate them.

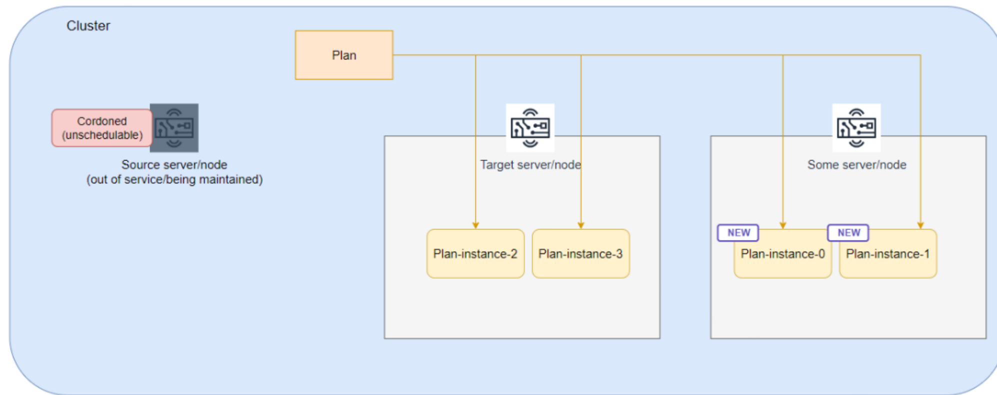



Figure 53. StatefulSet reschedules missing pods to another server from the resource pool.

5. Figure 54, StatefulSet is called again to scale Plan back down to its original instance number. The scaling down is done in reverse ordinal order, so in this case, the instances created at step 2, will be removed.

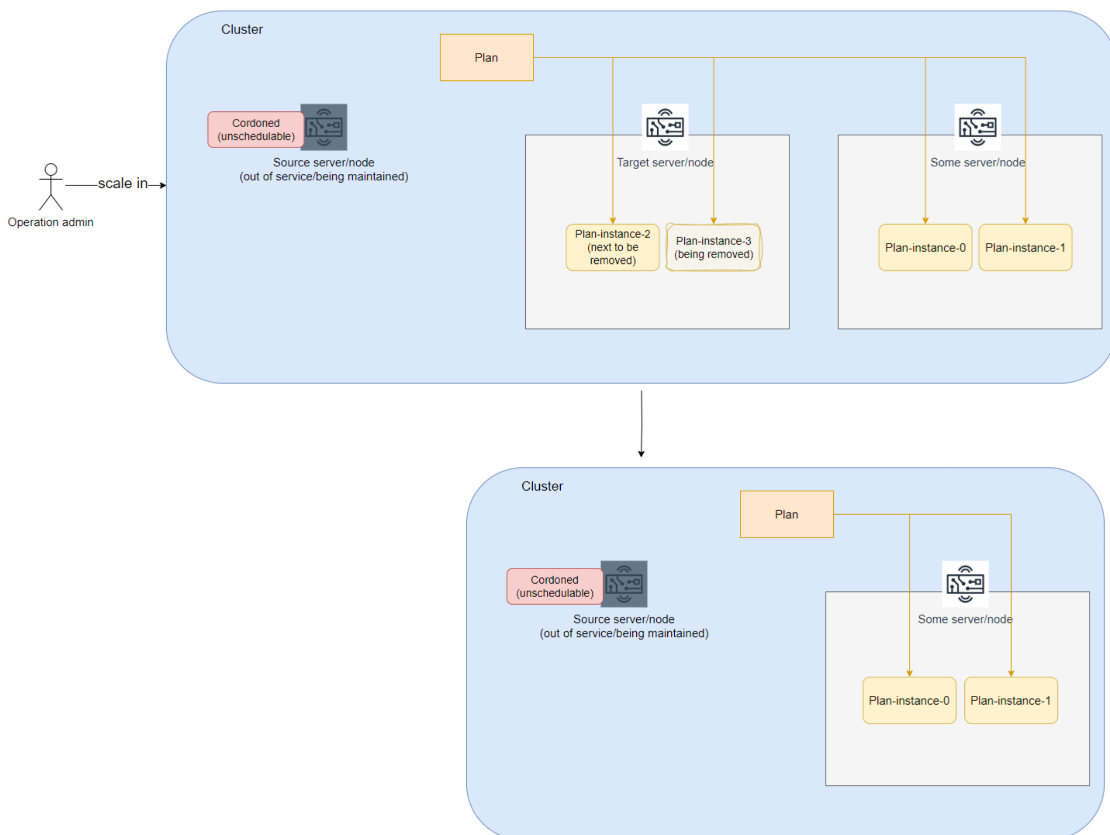



Figure 54. Scale back in as instances moved successfully.

6. Figure 55, eventually when the decommissioned server's maintenance is done, the server is uncordoned, so it can accept scheduling and is ready to be used again. The taint should also be removed if we'd like the server to be used again by the Plan in this scenario.

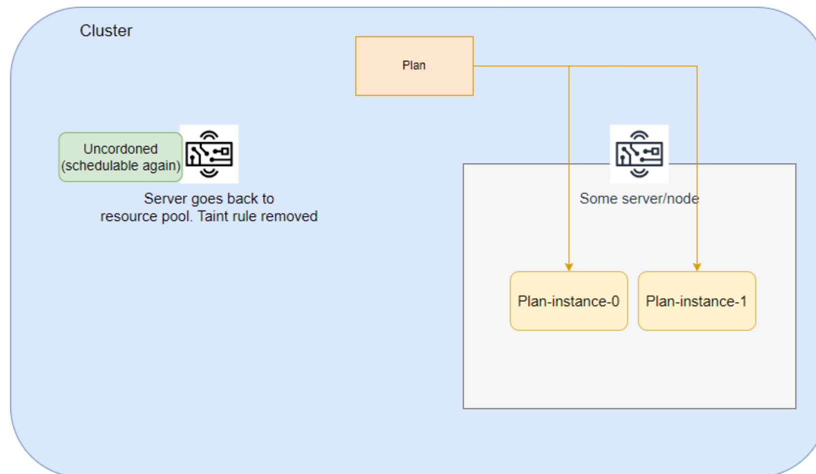



Figure 55. Server maintenance done, it is uncordoned and taint rule is removed.

### 8.5 Plan application update

As a stateful application, Plan update is a rather difficult topic. On the other hand, we want to make sure users don't experience downtimes when an application is being updated, and on the other hand, we want to ensure that the application works during and after the upgrade. Essentially non-disruption updates are quite hard to achieve, and could be a thesis topic of its own, but some higher-level design of zero downtime and minimal disruption can be thought of.

The initial problem to tackle here is that Plan experts would essentially want to use rolling updates with Kubernetes. However, how rolling updates with StatefulSets work is that it updates the pods one by one, this way it can be ensured that there will never be a case when all the pods are down at the same time, thus technically zero downtime is achieved. Some restrictions can be put that at least n-number of pods has to be always up, we will come back to this later in Subsection 8.6.1. Now, for the problematic part, as mentioned already in Section 6.3, Plan requires the other nodes to go OFFLINE so any migrations can be run for the new version, otherwise the database can become corrupted due to backwards incompatible changes.

Suggested solution for getting the rolling-updates is to make the database always backwards compatible with gradual updates or what can be also called additive migrations. The basic concept is illustrated very well in the following Figure 56.

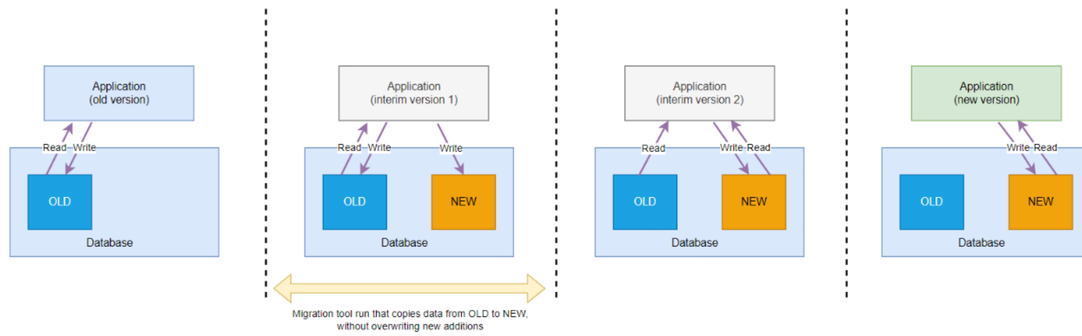



Figure 56. Introducing interim versions to achieve backwards compatible zero downtime version updates with rolling-updates. Adaption of (Ivon, 2023).

With these interim versions, we can maintain backwards compatibility and hence there is no need for the Plan to go offline for the migrations. The data migration starts when the interim version 1 is running and it should be ready by the time the final, or new version in the above figure, version is deployed. The interim version 2, where we still write to old schema, is needed precisely in the case of having rolling-update kind of strategy where portion of users are still seeing the previous interim version 1. Rolling update is a gradual update itself, but for the Kubernetes pods that are the Plan instances. It's important to understand that it's separate concept, albeit having some similarities, from database schema gradual update. The use cases described in the following subsections, with illustrations of both the user interface and database view, can help one understand the backwards compatibility.

It is good to mention here that if the interest is only getting non-disruptive database updates and migrations, Kubernetes is not a necessity. The backwards compatibility is prerequisite for using Kube's rolling-update, as users are using two different product versions at the same time (more of this in Subsection 8.6 Deployment Strategy). To get the backwards compatibility without rolling-update, the so called interim-versions can be ignored, but some kind of triggers need to be implemented that copies data from old to new schema before any insert, deletion or update happens (Ivon, 2023).

### 8.5.1 Adding a column to database schema

Figure 57 illustrates the first two phases of a gradual schema update. In the initial situation (Application version 1.0) we have a table with n-number of users. This table has a non-null constraint on both ID and Name (* indicates not null constraint). In the next version update (version 1.1), the user interface is still the same but an additional column "Email" has been added to the database. This new column accepts null values at this point which allows users, who still don't see the newly added column, to add a new person to the list.

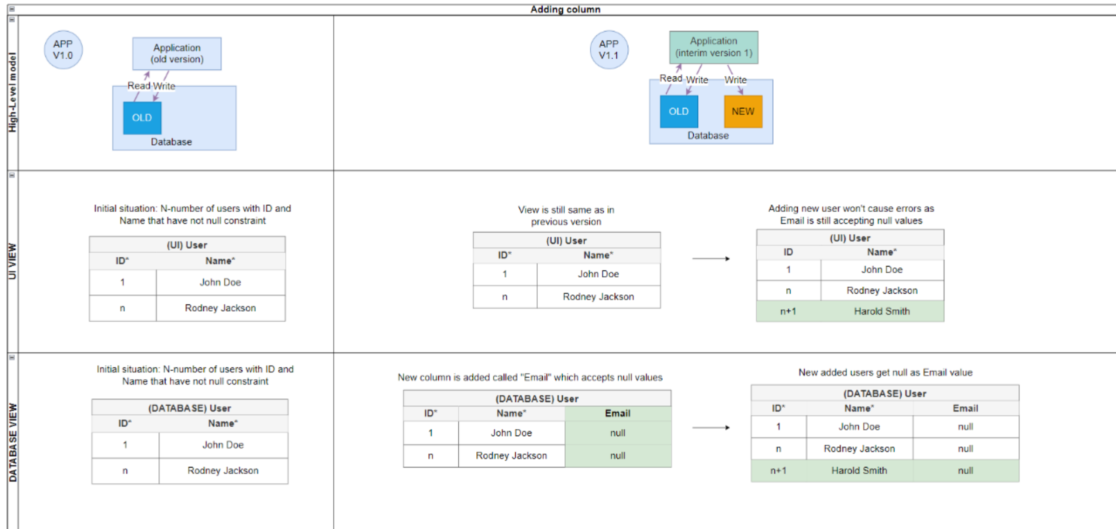



Figure 57. Adding a column Part 1.

In the next interim version show in Figure 58, the user interface is updated to have the new “Email” column, values matching with what the database has. Now the application users are able to modify the email field as well. A tool is run after this version update to fill the null values with some default value, in this illustrated case it is placeholder@email.com. In the “final” version, the read from the old database is removed and non-null constraint has been given to the Email field. Final version is fully reliant on the new schema.

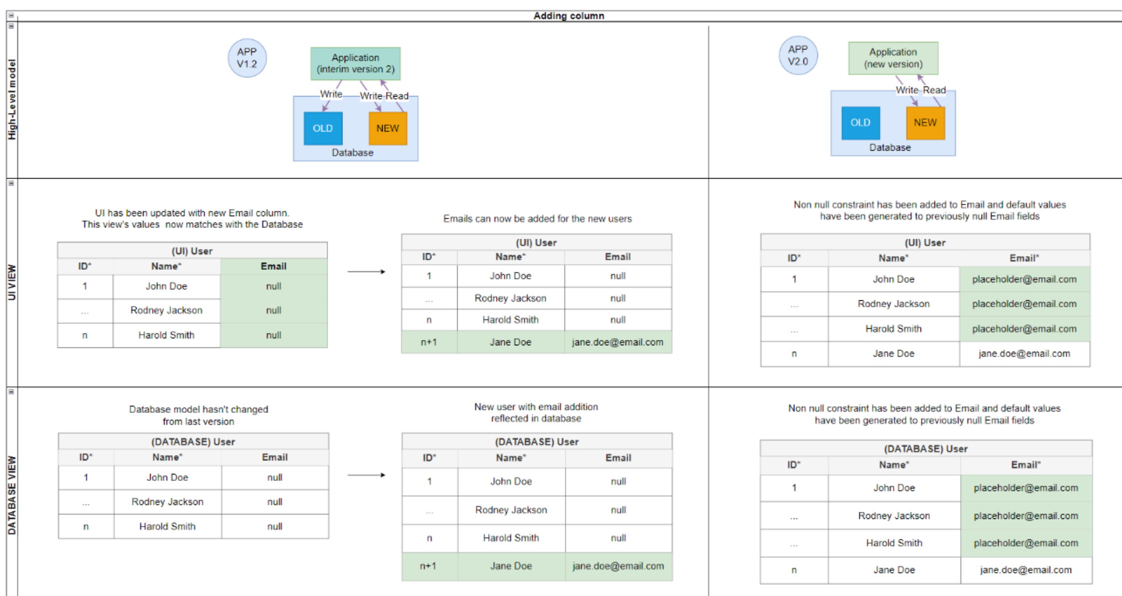



Figure 58. Adding a column Part 2.

### 8.5.2 Deleting a column from database schema

Another example is when something from a schema is deleted. In the illustrated case of Figure 59, it depicts Email column deletion and the first interim version of that process. The initial situation is basically where we left off in the previous demonstration when we added the column. There's now a table "User" with columns ID, Name and Email, all which have non-null constraints. The table has an n-number of separate users. For the first interim version, the non-null constraint is removed from the Email column on the database side, so we allow the Email field to be null for new user additions.

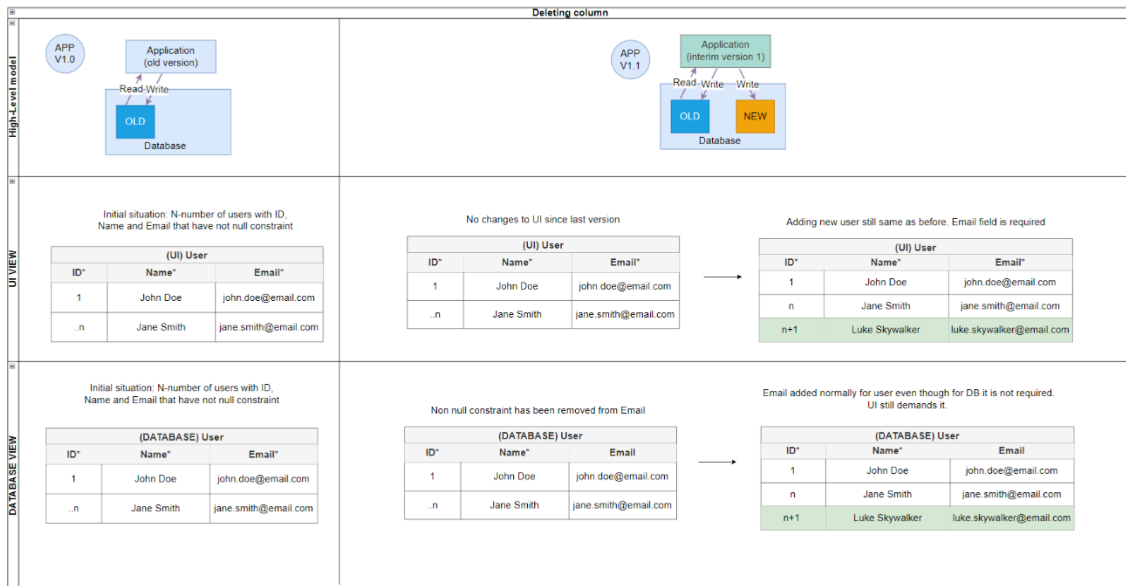



Figure 59. Deleting a column Part 1.

In the interim version 2, show in Figure 60, the whole Email column is removed from the UI side. At this point, when adding new users the Email value will receive null in the Database side (which is accepted) as users won't be able to fill the Email value on the UI anymore. In the final version, the Email column is finally removed from the Database as well.

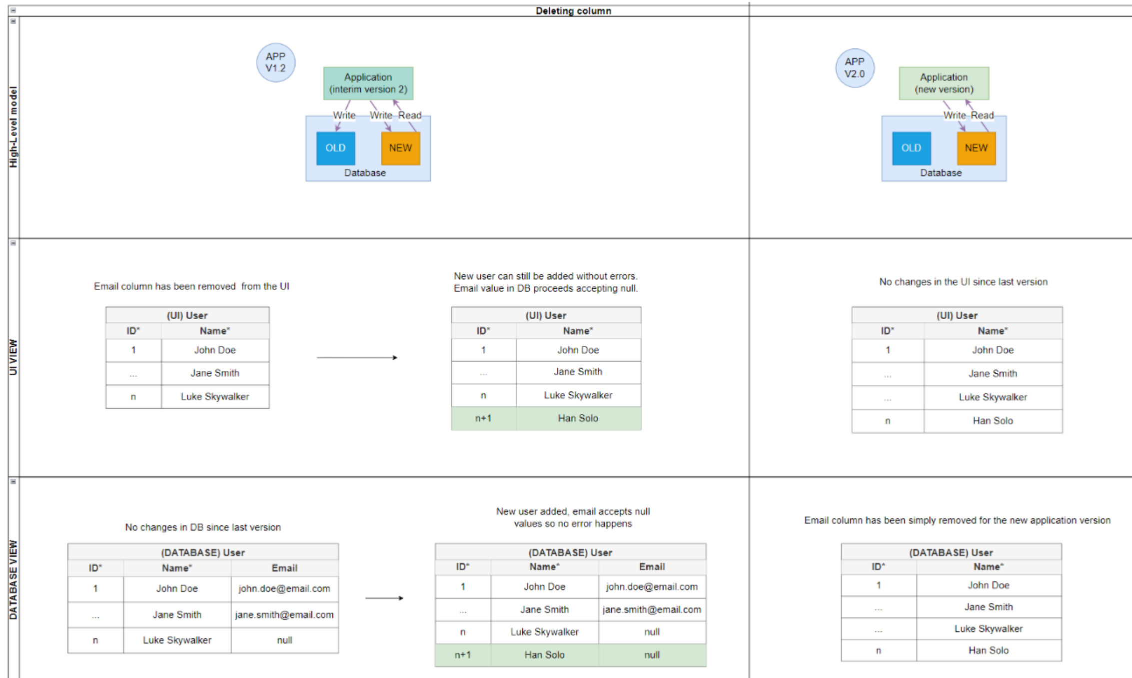



Figure 60. Deleting a column Part 2.

### 8.5.3 Renaming a column in database schema

This use case is quite classically depicted when thinking of additive migrations. There exists a field “Name” in the current/old application version, and we want to rename it so it would actually contain the “Firstname” and then another added field would contain “Surname”. Instead of modifying the Name field, we should just add new fields Firstname and Surname. While at first, we have three fields in parallel (Name, Firstname and Last-name), it helps us to support backwards compatibility and thus ensure availability for users of the old system.

Again, the initial situation depicted in below figure Figure 61: There’s a table of users, with ID and Name fields, both with non-null constraints. In the interim version 1, both “Firstname” and “Surname” columns will be added, both of which still accept null values. Any new user additions can already contain these new values.

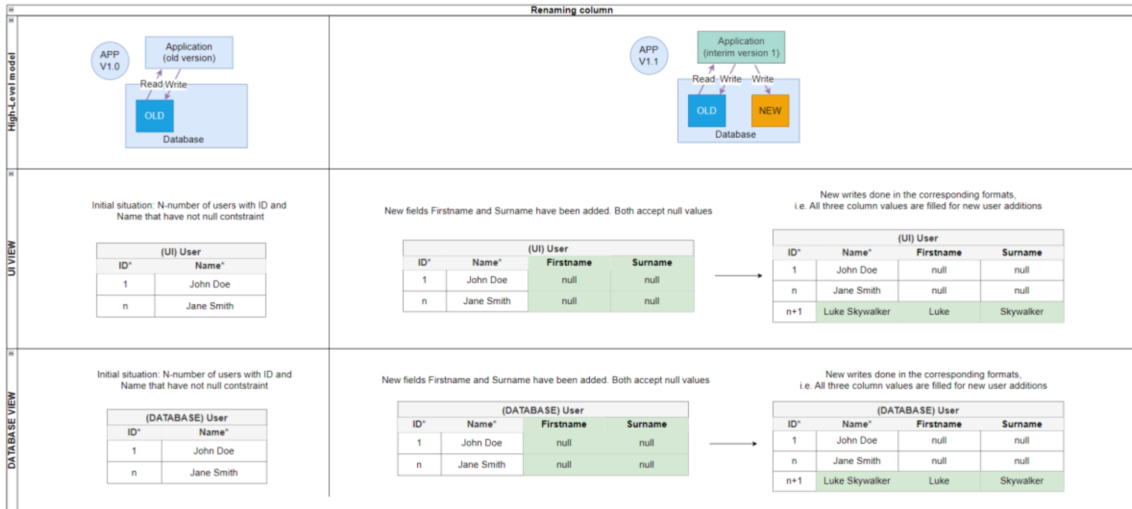



Figure 61. Renaming a column, Part 1.

For the interim version 2, Figure 62, the non-null constraint is removed from the “Name” field; any new additions to the table can skip the Name value (providing null). The migration tool populates the values for the new columns “Firstname” and “Lastname”, by referring to the “Name”-column. For the final version, the “Name”-column is removed from both UI and database, as there’s no need for it anymore.

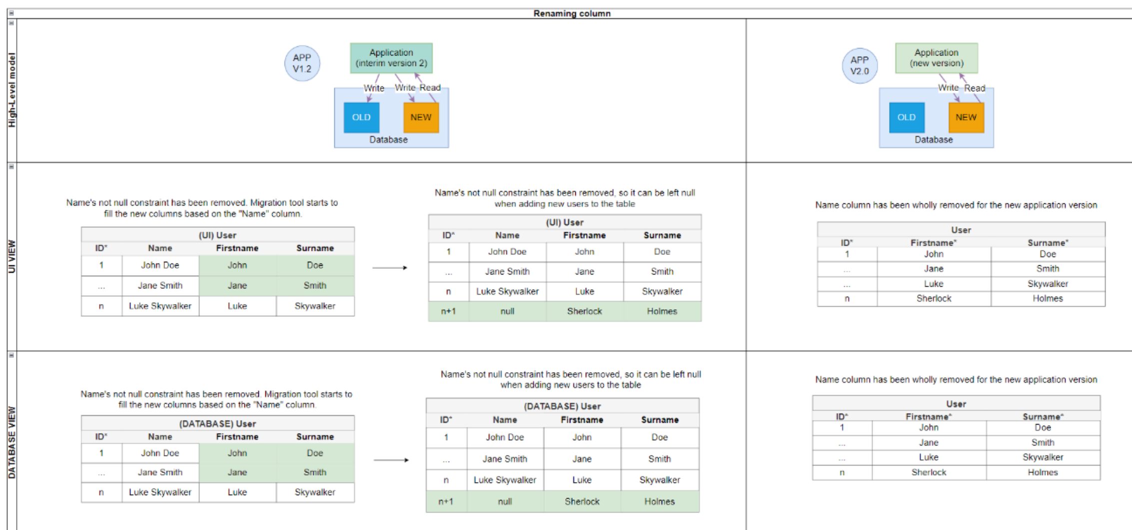



Figure 62. Renaming a column, Part 2.

### 8.5.4 Rollback

While the gradual upgrade allows backwards compatibility, rolling back essentially still causes data loss. With additive migrations, theoretically, there shouldn’t be a need to do actual rollback in the database. If we think about banking and bank transactions, one

doesn’t exactly do rollbacks to the previous state, but instead adds actions to reach a similar state as previously, this is the core idea of additive transactions or migrations. There was already an example of this in the earlier use cases. The initial state of the use case of adding a new column, Section 8.5.1 Adding a column, was the end state of the use case of deleting a column, Section 8.5.2 Deleting a column, we just did the actions backwards. If for some reason there’s still a need to manually set the database into its previous state, it should be accepted side-effect that there will be data loss happening.

## 8.6 Deployment Strategy

While not exactly requested by the commissioner, when researching about zero downtime upgrades, the deployment and release strategies naturally came up. So far, there have been mainly considerations to get the rolling update deployment strategy, when moving to use Kubernetes orchestration. As it has been established, continuous availability and stable performance throughout the update are important for customers, which means that we must scale out by the number of existing instances (briefly mentioned in Subsection 8.4) and then scale back in after the updates are done.

Canary deployment could also be considered as well since it seemed that feature testing and release quality were big topics and worries. Canary would allow testing functionality with a small group of actual end-users. I will be describing how rolling update and Canary would work in Kubernetes from Plan’s point of view.

### 8.6.1 Rolling update

In the rolling update strategy, the pods themselves are gradually updated (replaced) to the next version. Note here that this is a slightly different gradual update concept than what was explained in the previous section, where it focused on gradually updating version A to version B with intermediate versions in between.

The rolling update in Kubernetes basically means that we update the pods step-by-step, a certain amount of pods at a time. And with StatefulSet, the update always starts in reverse ordinal number; Pod with the highest number is deleted first and re-created with the new template. Kuberenetes’ default pod management policy is “OrderedReady”, which means that the pods are both deleted and updated one by one, as depicted in following Figure 63.

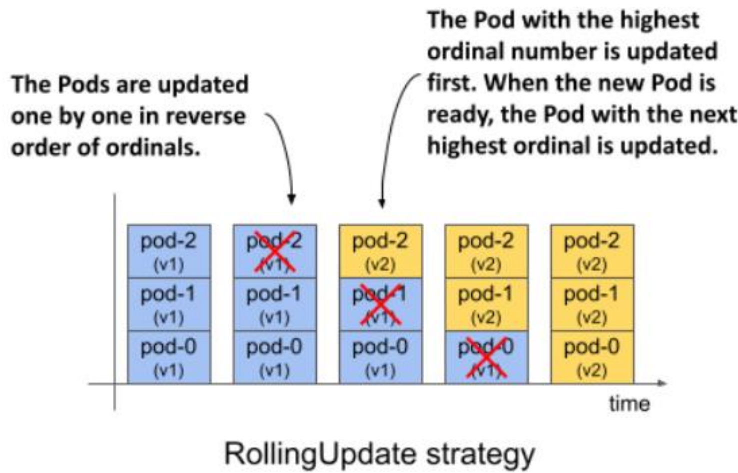



Figure 63. RollingUpdate strategy in Kubernetes for StatefulSets (Lukša, 2023).

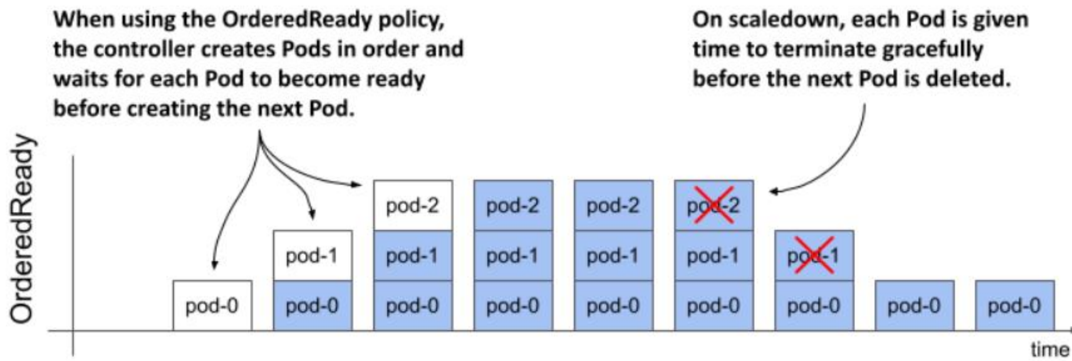



Figure 64. Default rolling update strategy with OrderedReady pod management policy. Pods are updated one by one in reverse ordinal order (Lukša, 2023).

OrderedReady -strategy, show in Figure 64, is good if there are dependencies between the pods, so it would be necessary to make sure the previous instance is ready before starting the setup of the next one. In other words, the pod updating would be done sequentially. For Relex Plan, the Plan roles are dependent on each other, but those are managed by Helix so the pod manager doesn't have to handle that. OrderedReady -strategy would be quite slow for Plan to use, as each pod would still have to go through the graceful shutdown, which as mentioned previously, might take days in the worst scenario Following Figure 65 shows the timeline view of regular OrderedReady rolling update with the scale out and scale in needs of Plan for keeping the performance and availability stable.

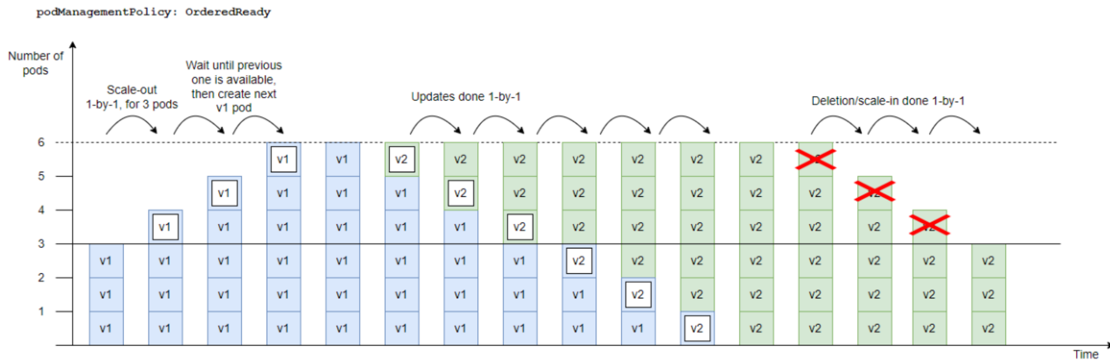



Figure 65. Timeline of rolling update with OrderedReady policy.

We can make the update slightly faster by enabling the "MaxUnavailableStatefulSet" feature flag and specifying the "maxUnavailable" property value. Like its name hints already, it specifies how many pods are allowed to be unavailable at once. With this property, one can control the availability and in a way the speed of the updates. In this particular use case, we are scaling out 100% in order to use the maxUnavailable to speed up the updates as it will allow us to update multiple pods in parallel while keeping the same performance as normally. In below Figure 66, the setup is the same as previously, but now with maxUnavailability set as 50%. The scale-out and scale-in still happen one-by-one but Kube will now update half the pods in parallel at once.

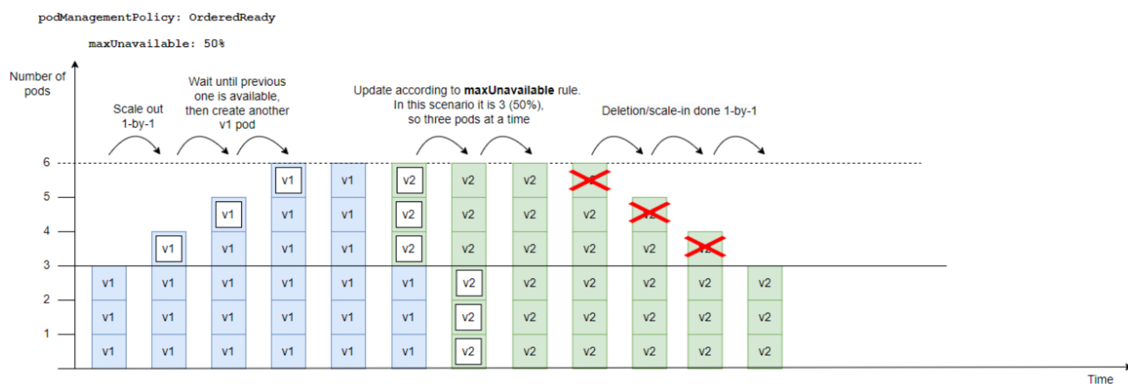



Figure 66. Timeline of rolling update with OrderedReady policy and maxUnavailable property. Pods version update happens in parallel while adhering to the maxUnavailable rule.

With maxUnavailable, we can use some parallelization during pod updates, which helps us save some time. The scale-out and scale-in phases are however still done one-by-one in the sequential model, which still takes quite a lot of time. For StatefulSets, there's another option pod management policy, which is "Parallel".

> Parallel pod management tells the StatefulSet controller to launch or terminate all Pods in parallel, and to not wait for Pods to become Running and Ready or

> completely terminated prior to launching or terminating another Pod. This option only affects the behavior for scaling operations. Updates are not affected. (Kubernetes D, 2023)

This allows Kube to do scaling in parallel as well, it only affects the scaling operation, which is why it should be used together with the maxUnavailable property if one wants updates to be done in parallel as well. With Parallel pod management policy and maxUnavailable property, it is possible to cut time-to-ready length, as represented in the bottom graph in Figure 67.

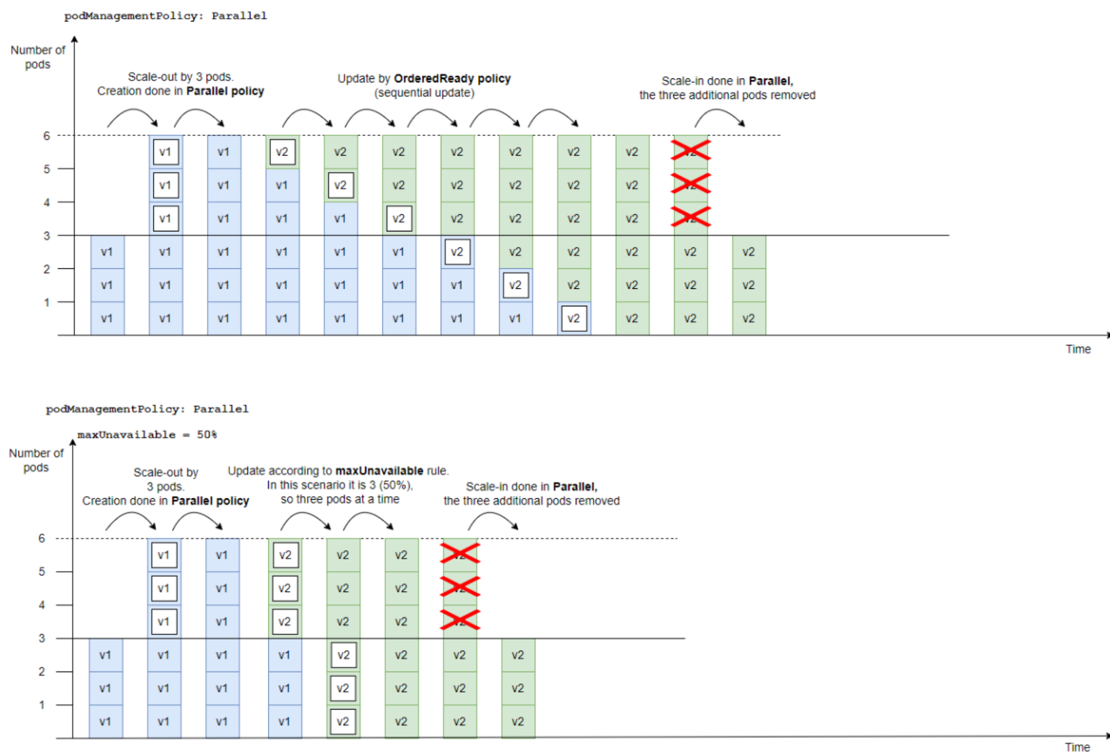



Figure 67. Timeline of rolling update with Parallel policy. The policy only affects the scaling operations, so the pod update (top) still happens sequentially. Together with the maxUnavailable property (bottom), we can make both scaling and updating happen in parallel.

### 8.6.2 Partitioned rolling update (Canary)

Canary update strategy could be considered if one wants to first test the reliability and quality of the new product version with a smaller subset of users before releasing it fully. The name of the Canary update comes from how miners used to use birds to help them alert of toxic gas.

> The name for this technique originates from miners who would carry a canary in a cage down the coal mines. If toxic gases leaked into the mine, it would kill the canary before killing the miners. A canary release provides a similar form of

early warning for potential problems before impacting your entire production infrastructure or user base. (Sato, 2014)

What happens is that the application traffic is split between the older version and the newly deployed version. The split percentage can be controlled and thus allowing a type of rolling update. In Kubernetes, they call it “Partitioned rolling update”, see Figure 68. With StatefulSets’ rolling update, it is not possible to pause the deployment but by its partition parameter, the StatefulSet can be split into two partitions that can be updated separately.

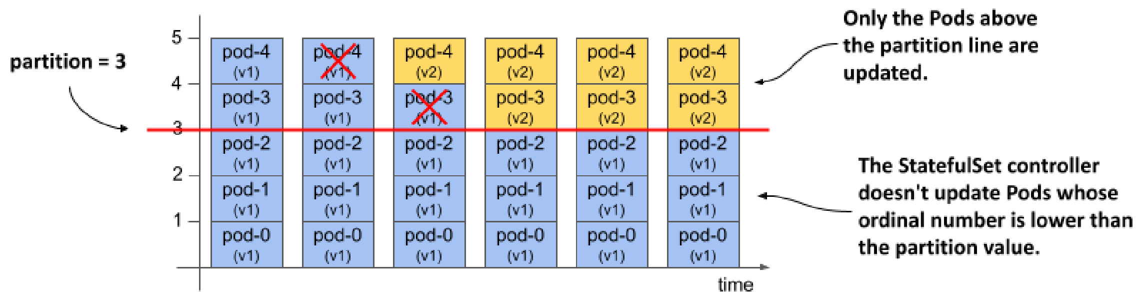



Figure 68. Timeline of partitioned rolling update aka. Canary deployment strategy. Allows more control over updates (Lukša, 2023).

For Relex Plan, that also needs the gradual version update itself to make the zero downtime migrations possible, the Canary deployment would look something like the following depiction in Figure 69. The backwards compatibility with partitions is shown in Figure 70.

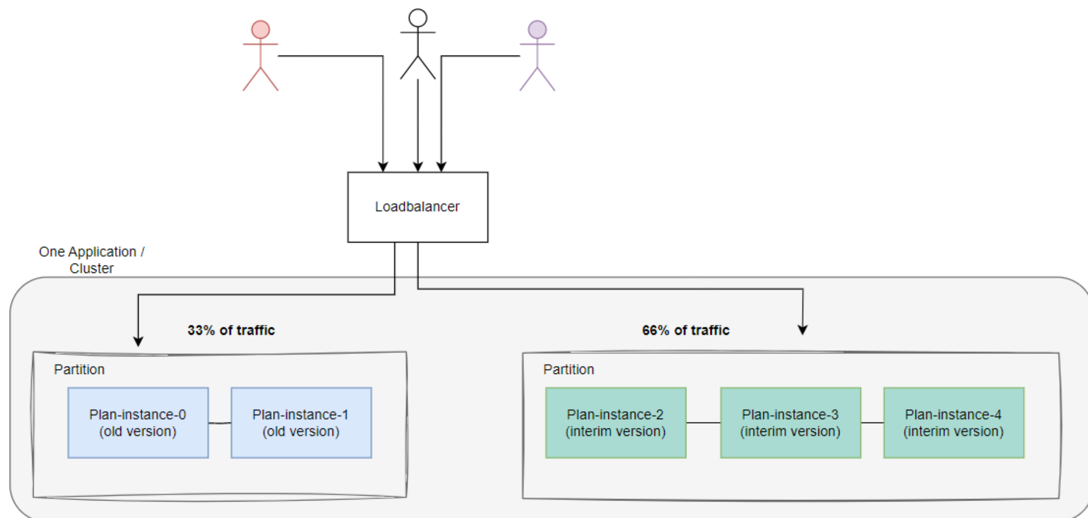



Figure 69. Relex Plan Canary deployment with 5 instances partitioned in two.

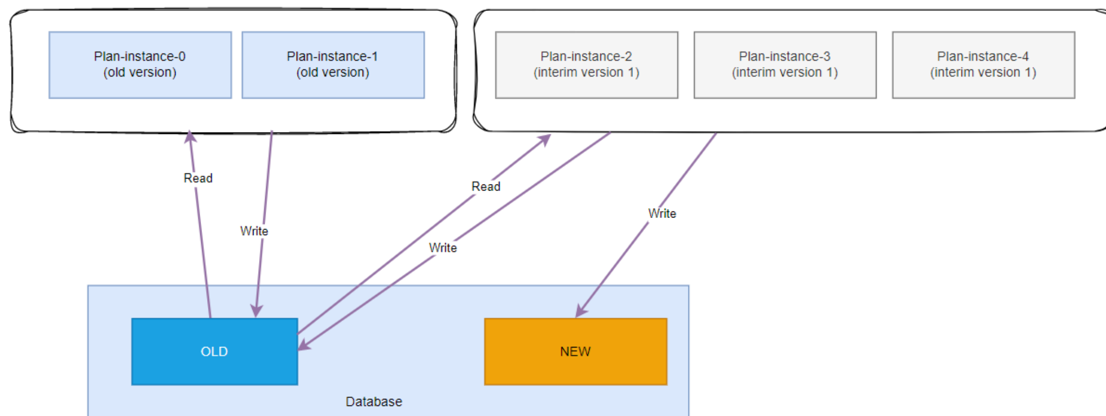



Figure 70. Depiction of how the backwards compatibility is kept with Canary deployment. The users that are routed to the old version can only interact with the old database schema, while the other user group using the newer version is already writing to the new one.

When used properly, the Canary model can be an efficient way to test out new features with actual end users while making sure there's an easy way to rollback. Basically, it's the same mindset as having alpha and beta version testers. The drawback is however that it will probably cause performance issues as we are splitting the resources. One idea would be to scale-out before partitioning, but it's a topic of its own of how long one would want to support two versions in parallel. Perhaps Canary could be used sparingly, only to test out major changes and features, but then this comes back to the point of using Canary if you are not really using it.

# 9 Conclusion

Kubernetes lifecycle management would improve Relex Plan’s lifecycle management by providing a good platform for it with its resource pooling capabilities. Deployment strategies like rolling update and canary can also be handled by Kube, and either of these strategies would be beneficial for the continuous integration and delivery or continuous deployment if that’s ever wanted. The automated lifecycle management and cluster orchestrations both benefits and provides well to the “fail fast and shift left” DevOps ideology.

The product would however need to improve for Kubernetes orchestration to have a positive return of investment, whether we are looking at monetary numbers or developer hours needed for the setup work. This thesis didn’t even take into account all the networking and security-related issues that would be needed to solve for Plan to work acceptably in Kube.

The main issue to be solved would be getting Plan schema updates backwards compatible, so version upgrades and updates can be done gradually. With backwards compatibility, the Plan upgrades can truly be non-disruptive for customer user experience, which should be one of the highest priorities along with high availability and eventually targeting for fault-tolerant system. To solve these issues, Kubernetes is not a must, the company could develop their own in-house PaaS system from scratch to fill their lifecycle and orchestration needs. Plan’s backwards compatibility is also basically a prerequisite to taking Kubernetes’ natively supported deployment strategies into use. But I think that the work for getting in-house lifecycle management and cluster orchestration as good as what we could achieve with Kubernetes, would most likely require too much effort. It seems that the architect experts in Relex are looking into containerizing Plan regardless, so Kubernetes most likely would come into the picture anyways in that case as it is the software industry’s de-facto tool for container deployment and orchestration. However, Plan’s particular need for an in-memory database, which has significant performance benefits compared to a split database, makes it hard to transform from monolithic to microservice architecture. The database split, to model microservice architecture, is an ongoing discussion in the company.

To generalize that the ideal ultimate goal is to have a continuous deployment with automated version upgrades and autoscaling; In Plan’s case, prerequisite being the database

schema’s backwards compatibility for zero downtime experience, building and supporting the deployment strategy and orchestration with Kubernetes would be able to fulfil the need smooth operation needs.

## References

Apache Helix, 2023. *Architecture.* [Online]
Available at: https://helix.apache.org/Architecture.html
[Accessed 27 02 2023].

Apache ZooKeeper, 2023. *Apache ZooKeeper.* [Online]
Available at: https://zookeeper.apache.org
[Accessed 27 02 2023].

Atlassian, 2023. *Microservices vs. monolithic architecture.* [Online]
Available at: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith
[Accessed 03 06 2023].

Barlett, J., 2023. *Cloud Native Applications with Docker and Kubernetes : Design and Build Cloud Architecture and Applications with Microservices, EMQ, and Multi-Site Configurations.* Berkeley: Apress.

Belmont, J.-M., 2018. *Hands-On Continuous Integration and Delivery.* Packt Publishing.

Briski, K. A., Chitale, P., Hamilton, V., Pratt, A., Starr, B., Veroulis, J., & Villard, B., 2008. *Minimizing code defects to improve software quality and lower development costs*. IBM.

Casalicchio, E. & Iannucci, S., 2020. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and computation,* 32(17), pp. 3-5.

Conway, M. E., 1968. *How Do Commitees Invent?.* [Online]
Available at: https://www.melconway.com/Home/Committees_Paper.html
[Accessed 14 02 2023].

Daley, S., 2022. *Company Culture.* [Online]
Available at: https://builtin.com/company-culture
[Accessed 24 1 2023].

Docker, 2023. *Why Docker.* [Online]
Available at: https://www.docker.com/why-docker/
[Accessed 19 02 2023].

Duvall, P. M., Glover, A. & Matyas, S., 2007. *Continuous Integration : Improving Software Quality and Reducing Risk.* 1st Edition. Addison Wesley.

ElGherani, N. S., 2022. Microservices vs. Monolithic Architectures. *Al-Mansour Journal,* 37(1), pp. 27-44.

Falck, M., 2013. *Big Data – Big Talk or Big Results?.* [Online]
Available at: https://www.relexsolutions.com/resources/big-data-big-talk-or-big-results/
[Accessed 15 06 2023].

Fogarty, K., 2012. *Where did 'cloud' come from?.* [Online]
Available at: https://www.computerworld.com/article/2726701/where-did--cloud--come-from-.html
[Accessed 08 02 2023].

Fowler, M. & Lewis, J., 2014. *Microservices.* [Online]
Available at: https://martinfowler.com/articles/microservices.html
[Accessed 14 02 2023].

Freeman, E., 2019. *DevOps for Dummies*. Newark: Wiley.

Getz, A., 2021. *Computer Hardware Abstraction: Virtual Machines vs Containers.* [Online]
Available at: https://bi-insider.com/posts/virtual-machines-vs-containers/
[Accessed 15 02 2023].

Google Cloud A, 2023. *Container Orchestration.* [Online]
Available at: https://cloud.google.com/discover/what-is-container-orchestration
[Accessed 19 02 2023].

Google Cloud B, 2023. *What is object storage.* [Online]
Available at: https://cloud.google.com/learn/what-is-object-storage
[Accessed 06 03 2023].

Google Cloud: Cloud Architecture Center, 2023. *DevOps culture: Westrum organizational culture.* [Online]
Available at: https://cloud.google.com/architecture/devops/devops-culture-westrum-organizational-culture
[Accessed 17 07 2023].

Gopalakrishna, K., Lu, S., Zhang, Z., Silberstein, A., Surlaker, K., & Subramonian, R. S., 2012. *Untangling Cluster Management with Helix.* ACM, pp. 1-13.

Haloi, S., 2015. *Apache ZooKeeper essentials : a fast-paced guide to using Apache ZooKeeper to coordinate services in distributed systems.* 1st edition. Birmingham: Packt Publishing.

Hevner, A. R., March, S. T., Park, J. & Ram, S., 2004. *Design Science in Information Systems Research.* [Online]
Available at:
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.1725&rep=rep1&type=pdf
[Accessed 06 09 2023].

Hewlett Packard, 2017. *Agile is the New Normal: Adopting Agile Project Management.* [Online]
Available at: https://softwaretestinggenius.com/docs/4aa5-7619.pdf
[Accessed 23 1 2023].

Hightower, K., Burns, B. & Beda, J., 2017. *Kubernetes: Dive into the Future of Infrastructure.* O'Reilly.

IBM, 2023. *Kubernetes.* [Online]
Available at: https://www.ibm.com/topics/kubernetes
[Accessed 28 02 2023].

Isenbeg, K., 2017. *Container Orchestration Wars.* [Online]
Available at: https://learning.oreilly.com/videos/velocity-2016/9781491944646/9781491944646-video255636/
[Accessed 19 02 2023].

Ivon, Y., 2023. *Upgrading database schema without downtime.* [Online]
Available at: https://blog.devgenius.io/upgrading-database-schema-without-downtime-9961070b9016#3cec
[Accessed 13 04 2023].

Kebbani, N., McKendrick, R. & Tylenda, P., 2022. *The Kubernetes Bible.* Packt Publishing.

Kleppman, M., 2017. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly.

Kobilinskiy, A., 2019. *Cloud Service Models: SAAS, PAAS, IAAS - Which Is Better For Business.* [Online]
Available at: https://dev.to/artemkobilinskiy/cloud-service-models-saas-paas-iaas-which-is-better-for-business-574k
[Accessed 08 02 2023].

Kubernetes A, 2023. *Glossary.* [Online]
Available at: https://kubernetes.io/docs/reference/glossary/
[Accessed 28 02 2023].
Kubernetes B, 2023. *Topology Spread Constraints.* [Online]
Available at: https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/
[Accessed 05 09 2023].

Kubernetes C, 2023. *Taint and toleration.* [Online]
Available at: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/
[Accessed 05 09 2023].

Kubernetes D, 2023. *StatefulSets.* [Online]
Available at: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/
[Accessed 12 04 2023].

Kubernetes E, 2023. *Docs.* [Online]
Available at: https://kubernetes.io/images/docs/kubectl_drain.svg
[Accessed 09 09 2023].

Kubernetes, 2023. *Components.* [Online]
Available at: https://kubernetes.io/docs/concepts/overview/components/
[Accessed 28 02 2023].

Liker, J. &. R. K., 2016. *The Toyota Way to Service Excellence: Lean Transformation in Service Organizations.* 1st edition. McGraw-Hill.

Lukša, M., 2018. *Kubernetes in action.* 1st edition. Manning Publications.

Lukša, M., 2023. *Kubernetes in action second edition.* [Online]
Available at: https://livebook.manning.com/book/kubernetes-in-action-second-edition/chapter-15/v-15/323
[Accessed 02 06 2023].

Lukša, M., 2023. *Kubernetes in action second edition.* [Online]
Available at: https://livebook.manning.com/book/kubernetes-in-action-second-edition/chapter-15/v-15/251
[Accessed 21 08 2023].

Lukša, M., 2023. *Kubernetes in action second edition.* [Online]
Available at: https://livebook.manning.com/book/kubernetes-in-action-second-edition/chapter-15/v-15/296
[Accessed 06 02 2023].

McGee, J., 2016. *The 6 steps of the container lifecycle.* [Online]
Available at: https://www.ibm.com/blogs/cloud-computing/2016/02/08/the-6-steps-of-the-container-lifecycle/
[Accessed 19 02 2023].

Mukherjee, J., 2023. *Business Value.* [Online]
Available at: https://www.atlassian.com/continuous-delivery/principles/business-value
[Accessed 07 02 2023].

Nadareishvili, I., Mitra, R., McLarty, M. & Amundsen, M., 2016. *Microservice Architecture: aligning principles, practices, and culture.* 1st Edition. Beijing: O'Reilly.

Newman, S., 2021. *Monolith to Microservices.* Upfron Books.

Nikhil, S., 2020. *CALMS Framework, A Successful DevOps Model.* [Online]
Available at: https://nikhils-devops.medium.com/calms-framework-a-successful-devops-model-d471af67149b
[Accessed 30 01 2023].

NIST, 2011. *The NIST Definition of Cloud Computing.* [Online]
Available at: https://csrc.nist.gov/publications/detail/sp/800-145/final
[Accessed 08 02 2023].

Patel, A., 2021. *Kubernetes — Architecture Overview.* [Online]
Available at: https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd
[Accessed 28 02 2023].

Patel, A., 2021. *Kubernetes — Services Overview.* [Online]
Available at: https://medium.com/devops-mojo/kubernetes-services-overview-k8s-service-introduction-why-and-what-are-kubernetes-services-how-works-e6fd4fd4a51a
[Accessed 01 03 2023].

Philippe, M., 2023. *Kubernetes Programming with Go : Programming Kubernetes Clients and Operators Using Go and the Kubernetes AP.* Berkeley: Apress.

Piedad, F. & Hawkins, M., 2001. *High availability : design, techniques, and processes.* 1st edition. Prentice Hall PTR.

Pittet, S., 2023. *Continuous integration vs. delivery vs. deployment.* [Online]
Available at: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment
[Accessed 07 02 2023].

Polencic, D., 2020. *Graceful Shutdown.* [Online]
Available at: https://learnk8s.io/graceful-shutdown
[Accessed 12 04 2023].

Red Hat, 2020. *Stateful vs stateless.* [Online]
Available at: https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless
[Accessed 12 04 2023].

Regaldo, A., 2011. *Who Coined 'Cloud Computing'?.* [Online]
Available at: https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/
[Accessed 08 02 2023].

Richardson, C., 2023. *Pattern: Monolithic Architecture.* [Online]
Available at: https://microservices.io/patterns/monolithic.html
[Accessed 14 02 2023].

Rossel, S., 2017. *Continuous integration, delivery, and deployment : reliable and faster software releases with automating builds, tests, and deployment.* 1st Edition. Packt.

Roundtree, D., Castrillo, I. & Jiang, H., 2014. *The Basics of Cloud Computing : Understanding the Fundamentals of Cloud Computing in Theory and Practice.* 1st Edition. Amsterdam: Syngress, an imprint of Elsevier.

Ruparelia, N., 2016. *Cloud Computing.* The MIT Press.

Salimi, S., 2023. *Fail Fast.* [Online]
Available at: https://www.agile-academy.com/en/agile-dictionary/fail-fast/
[Accessed 31 01 2023].

Sato, D., 2014. *CanaryRelease.* [Online]
Available at: https://martinfowler.com/bliki/CanaryRelease.html
[Accessed 02 06 2023].

Schenker, G. N., Saito, H., Lee, H.-C. C. & Hsu, K.-J. C., 2019. *Getting Started with Containerization: Reduce the Operational Burden on Your System by Automating and Managing Your Containers.* Birmingham: Packt Publishing.

Shejwal, M., 2022. *Difference between virtualization and containerization.* [Online]
Available at: https://www.tutorialspoint.com/difference-between-virtualization-and-containerization
[Accessed 15 02 2023].

Steven, J., 2018. *What's the difference between agile, CI/CD, and DevOps?.* [Online]
Available at: https://www.synopsys.com/blogs/software-security/agile-cicd-devops-difference/
[Accessed 07 02 2023].

Van Steen, M. & Tanenbaum, A. S., 2017. *Distributed systems.* 3rd Edition.

Weil, S., Brandt, S., Miller, E. & Maltzahn, C., 2006. *CRUSH: controlled, scalable, decentralized placement of replicated data.* ACM, pp. 11-17.

Westrum, R., 2004. A typology of organizational cultures. *BMJ Quality & Safety,* Volume 13, pp. ii22-ii27.

Wiedemann, A. et al., 2019. *The Devops Phenomenon.* [Online]
[Accessed 30 1 2023].

Yıldırım, A., 2019. *DevOps Lifecycle: Continuous Integration and Development.* [Online]
Available at: https://medium.com/t%C3%BCrk-telekom-bulut-teknolojileri/devops-lifecycle-continuous-integration-and-development-e7851a9c059d
[Accessed 04 09 2023].

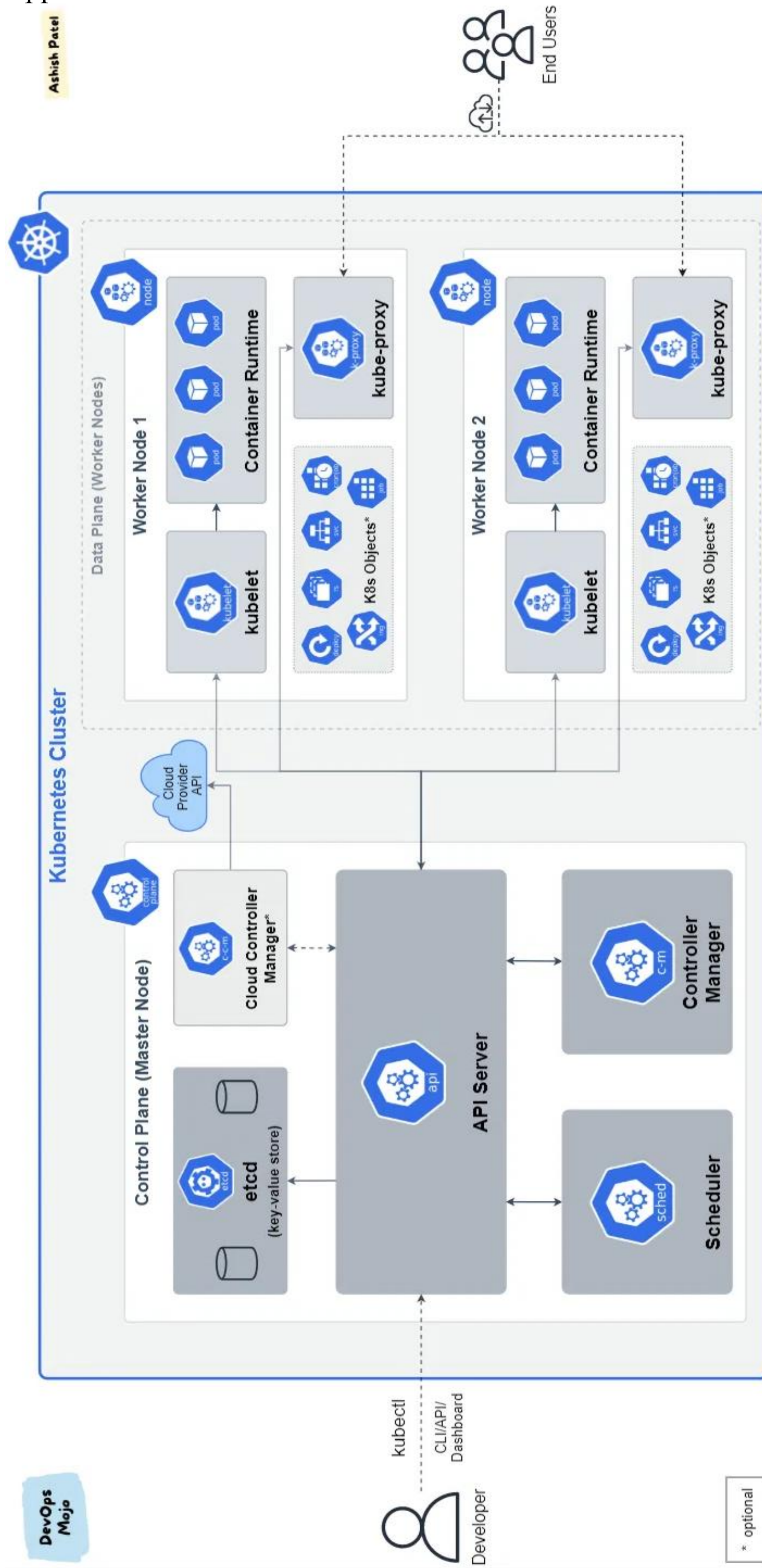
Yoshinori, M., 2014. *Semi-Synchronous replication at facebook.* [Online]
Available at: http://yoshinorimatsunobu.blogspot.com/2014/04/semi-synchronous-replication-at-facebook.html
[Accessed 20 02 2023].

Zettler, K., 2023. *What is a distributed system?.* [Online]
Available at: https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture
[Accessed 20 02 2023].

ZooKeeper, 2022. *ZooKeeper.* [Online]
Available at: https://zookeeper.apache.org/doc/r3.2.2/zookeeperOver.html
[Accessed 06 03 2023].

# Appendices

Appendix A: Kubernetes Cluster Architecture



# Appendix B: Relex Plan Update Full Sequence Diagram