Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Best Practices of Cloud Native Application Development

Bachelor of Profession's Thesis

Author
Dávid Richárd Kertész

Advisors
dr. Károly Farkas
Gergely Szabó

May 23, 2021

SZAKDOLGOZAT FELADAT

Kertész Dávid Richárd
szigorló üzemmérnök-informatikus hallgató részére

Best Practices of Cloud Native Application Development

IT cloud (both public and private) became ubiquitous recently. Cloud technologies made it possible to roll out application releases quickly, scale the application capacity almost endlessly while reaching unprecedented availability. The most successful branch of cloud technologies is referred to as cloud native, where the application components are containerized and the containers are orchestrated by a central orchestrator. Cloud native techniques rely on automation to a large extent.

The candidate's assignment is to study the state-of-the-art cloud native tools and architectures, understand the related application development workflow and experiment with cloud native environments. Thus, in the frame of the bachelor thesis work, the following tasks have to be completed:

- Study the state-of-the-art cloud native tools and architectures;
- Develop a simple demo application that follows the microservices architecture and 12-factor principles, as it is stateless, containerized, etc;
- Implement a CI solution that automatically tests, builds container images of the application components;
- Implement a CD solution that automatically deploys the application components into the provided Kubernetes-based cloud platform;
- Integrate the application with the platform monitoring and logging solution;
- Perform measurements to validate and demonstrate the following workflow:
    - Trigger a faulty application alert via the platform monitoring solution;
    - Troubleshoot the application with the help of dashboards and collected logs;
    - Deploy the fixed version of the application.
- Document the results and learnings.

Egyetemi témavezető: Dr. Farkas Károly, egyetemi docens, BME-VIK HIT tanszék

Ipari konzulens: Szabó Gergely, szoftver mérnök, Origoss Solutions Ltd.

Budapest, 2021. február 15.

Dr. Imre Sándor
egyetemi tanár
tanszékvezető

Témavezetői vélemények:

Egyetemi témavezető: □ Beadható, □ Nem beadható, dátum: aláírás:

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

1117. Budapest, Magyar tudósok körútja 2. I. Ép. B. 121.
Telefon: 463-3261 Fax: 463-3263
URL: http://www.hit.bme.hu

HALLGATÓI NYILATKOZAT

Alulírott *Kertész Dávid Richárd*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, May 23, 2021

Kertész Dávid Richárd
hallgató

Contents

Kivonat

A felhőalapú számítástechnika fejlődése felgyorsította a felhőalapú megoldások integrálását a szoftverfejlesztési és üzemeltetési folyamatokba. Azon vállalatok, amelyeknek nincsenek meg a megfelelő erőforrásaik arra, hogy a föld körül bárhol biztosítsák szolgáltatásaikat, felhasználhatják a felhőplatformok által nyújtott igényszerinti számítási erőforrások lehetőségét. Ezáltal a kisebb vállalatok is versenyképesek maradnak a nagyobbakkal szemben, és nyereségesek is maradhatnak az üzemeltetési költségeik óvatos menedzselésével.

Ez a folyamat, amiben a vállalatok elmozdulnak a saját maguk által fenntartott szerverektől a felhőalapú rendszerek irányába, helyet adott több olyan technológiának és metodikának, amelyek jelenleg a szoftverfejlesztést és üzemeltetést alapjaikban meghatározzák. A DevOps egy ilyenfajta kulturális folyamat. Magában foglal több agilis gyakorlatot és metodikát, amelyek a modern szoftverfejlesztés részei, míg a nagyrétű automatizáció felgyorsította a szoftverek beüzemelését. Ezek összeségükben napi szintről órákra csökkentették a szoftver beüzemelési idejét, míg a minőségellenőrzés szempontjai ugyanúgy teljesülnek.

Az ezen gyakorlatok által meghatározott fejlesztési folyamatot is nagy mértékben automatizálták. A folytonos folyamatok (continuous practices) meghatározzák az IT csapatok napi munkavégzését. A csapatok gyors reagálási idejét részben a konténerizációs technológiák használata alapozza meg, mely által egy absztrakciós réteg kerül a szoftver és az azt futtató környezet közé. Ezáltal pedig a több konténerből álló rendszerek orkesztrációja fölé is lehet vonni egy absztrakciós réteget, amire az egyik bevett technológia a Kubernetes. A Kubernetest nagymértékben kezdték el használni, miután a Google nyílttá tette a forráskódját, és manapság vezető technológia a felhőalapú üzemeltetésben.

Az előbbiekben említett folyamatok a szoftverfejlesztésben és üzemeltetésben, valamint a fejlett felhőalapú technológiák részeit képezik annak, amit úgy hívunk, hogy cloud-native. A cloud-native ökoszisztéma egy szorosan összefüggő szövete több, egymással kapcsolatban álló technológiának, amiknek a célja, hogy nyílt forráskódú megoldásokat biztosítsanak a fejlesztési és üzemeltetési folyamatok tisztán felhőalapú rendszerbe való migrálására.

Ennek a szakdolgozatnak a célja ezen tényezők leírása. A dolgozat első része a Kubernetest írja le, a legalapvetőbb működését, és hogy milyen érdemben változtatta meg a számítástechnika világát. A második rész egy gyűjtemény a fontosabb fejlesztési és üzemeltetési metodikákból és gyakorlatokból, melyeket a modern IT csapatok az egyre inkább online világban való hatékonyságért és produktivitásért követnek. A harmadik rész bemutatja azt a Kubernetes alapokon működő, komplex cloud-native rendszert, melyben egy egyszerű, a leírt gyakorlatokat követő folyamattal fejlesztett alkalmazás fut, amelynek a célja, hogy tesztelhető legyen a működésén keresztül a felhőrendszer hatékonysága. Egy teszteset is leírásra kerül, mely példaként szolgál a teljes cloud-native rendszer működésére éles környezetben.

Abstract

The evolution of cloud computing has accelerated the adoption of cloud-based solutions for developing and operating software. Companies who might not have adequate resources to provide their services to customers worldwide can utilise the powers of the cloud to access computing resources on-demand. This enables smaller companies to provide competition to larger ones, and to stay profitable by carefully managing operations costs.

This shift from privately owned servers to easily accessible cloud platforms has brought on a set of new technologies and methodologies that define software development and operation today. DevOps is such a cultural shift. It entails many separate agile practices and methodologies that make up the modern development process, while extreme automation hastens the deployment of software. These combined push down deployment time of new software from days to hours, while keeping the same measure of quality control.

The deployment of software developed using these techniques has seen high amounts of automation. Continuous practices define how IT teams deploy software on a day-to-day basis. The quick response time of development teams is enabled in part by containerisation technologies that abstract away the exact operating environment from the application that is to be deployed. This means that the orchestration of multi-container systems can be abstracted away as well, and one of the main solutions for this is Kubernetes. Kubernetes has seen a large increase in usage after its open-source release by Google and today is one of the leading solutions for cloud-based operations.

The aforementioned shift in development and operations culture, and the evolution of cloud-based technologies form a part of what today is called being cloud-native. The cloud-native ecosystem is a tight-knit group of interconnected technologies whose purpose is to provide open-source solutions for the migration of software development and operations into a purely cloud-based environment.

This thesis aims to describe these aspects. The first part describes Kubernetes, how it operates on the most basic level, and gives an introduction into how it transformed the IT world. The second part provides a collage of important development and operations methodologies and practices that modern IT teams follow in order to be productive and efficient in an increasingly online world. The third part introduces a complex, cloud-native system based on Kubernetes that operates a rudimentary application designed using the detailed practices to test the efficiency of this system. A test case scenario will also be documented, showing how a completely cloud-native system can be run in a production environment.

Chapter 1

Introduction

The rapid evolution of cloud computing has completely changed the IT landscape. The spread of resourceful cloud providers and the development of an increasing number of open-source solutions has opened up the way for IT companies to achieve high-availability and worldwide spread while staying cost-effective and agile in their development processes. Cloud computing provides customers with on-demand computing resources, such as CPUs and memory, while cloud providers handle differing amount of operative tasks to maintain these resources, ranging anywhere from physical maintenance to operating system administration. Companies that otherwise might not be able to afford computing resources to provide their services thus can develop these services and rely on cloud providers for resources. The affordable pricing of these resources keep well-performing companies profitable, reinforcing this cycle as long as it remains the best course of financial planning.

In an increasingly online world, high-availability has become a critical operations requirement. Year-long availability, meaning that services are running and accessible by customers, can be measured in percentages. A difference between 98% and 99% availability is a single percentage point; however, it amounts to days in a year[1]. This kind of downtime is unacceptable in critical services and undesirable when services are provided around the globe, on-demand. This gives rise to two main questions: how to develop highly-available services, and how to operate them in a highly-available manner. Cloud computing provides the resources needed, only the development and operations tasks need to be accomplished. This thesis takes a detailed look at some of the practices and technologies to do that. In Chapter 2, an overview of the orchestration tool Kubernetes will be drawn regarding its use in different areas of IT and how it operates on its most basic level. After that, Chapter 3 will detail some of the basic development and operations practices that are in use in the industry to develop and maintain services in a highly-available and scalable environment. This entails a focus on microservices in Section 3.1 as one of the main design choices, the event sourcing design pattern in Section 3.2 for data handling in a distributed system, the basic design requirements of a web application in Section 3.3, and the tools and practices connected to the DevOps culture in Section 3.4. In Chapter 4, everything detailed previously will be demonstrated by using a rudimentary software solution, which is able to generate logs and metrics connected to its functions, inside a Kubernetes environment to generate feasible operating errors. These will be picked up by the monitoring solutions integrated into the system, and the practices detailed will be used to react in an agile manner and solve the underlying problems. Finally, in Chapter 5, conclusions will be drawn based on the experiences gathered in previous chapters.

[1] Considering a year as 365 days, 98% availability is 7.3 days without available services. At 99%, it is 3.65, a difference of more than 3 days.

Chapter 2

Kubernetes

In Chapter 2, Kubernetes will be introduced as the solution for orchestrating container-based application architectures. Kubernetes is one of the main solutions for handling the complete lifecycle of containerised software in cloud environments. It enables operators of cloud-native software solutions to dynamically carry out their operations tasks with increased automation. As Kubernetes has been used in this thesis, it is important to detail specific aspects of its operation.

2.1 Introduction

Kubernetes is a container orchestration solution, which provides multiple layers of abstraction over the individualised management of container-based application architectures. It provides tools and objects for the automated installation, operation and management of individual containers inside clusters. These make the dynamic scaling and load balancing of the maintained services possible.

The system inside Kubernetes is declaratively described by objects. Such objects are responsible for the management of Pods that comprise of containers, the networking fabric in place to establish communication between objects, and other auxiliary objects performing tasks such as certificate handling, proxying, load balancing or service discovery.

One of the advantages of declarative configuration is the possibility of "lazy automation". The intended state must be defined by the operator and passed to Kubernetes through its API server. One of the ways is by describing them in YAML files. The data in these files is interpreted by Kubernetes, and stored as declarations of all the resources that it must manage by provisioning compute resources to each object according to their specifications. The control process of Kubernetes constantly monitors its underlying infrastructure for the existence of these resources. If there is a difference between the intended state stored inside these declarations and the actual state of the infrastructure, Kubernetes initiates changes so that these two are converged in a process called reconciliation. For example, at the very first second a Pod object might not have any containers running, and slowly Kubernetes makes sure, that it resembles the intended state, with a fully set up container and its system resources.

This way, should the administrator apply changes to a system by updating the configuration, the intended state description is updated, and Kubernetes carries out the reconciliation all over again after noticing the differences between the declaration and the resource's actual state. Through this it is able to notice if there are no changes between the two,

and whether there is a need for reconfiguration. The object declarations in Kubernetes are only changed if a different specification is applied to them. Its idempotence makes it possible to apply batches of configurations while expecting updates to only be carried out where necessary. This makes versioning easy. For the sake of example, if all resources are considered versioned objects, applying changes causes all these objects to implement them and refresh their version numbers. However, not all might carry out any reconfiguration as they are already at the desired state (i.e. there was no change in their description) but the version still shows the most recent point. The whole batch can always be considered the latest configuration in a version control system, as opposed to individually applying each, with a high chance of human error and the possibility of losing track of the state at which each resource should be.

The layered approach, the open-source code and the availability of an API also makes it easy to make custom implementations of Kubernetes itself or its various services. IT teams can make self-servicing distributions for automated development and deployment. Operators can provide an easy-to-use interface for development teams to provision resources through the API, so that the operations team can focus on optimising and troubleshooting deep-lying issues and functions in the infrastructure. This means that the development and operations teams do not have to constantly interact as routine infrastructure tasks are automated.

2.2 Usage

Kubernetes has seen a big increase in usage since its version 1.0 release in 2015. It has gone through around quarterly to half-yearly updates since then and is at version 1.20 at the start of 2021 with its end of support date set at December 2021.

In JetBrains' 2020 developer survey [64], 60% of developers with knowledge in infrastructure development said they use Docker as a server templating tool. Out of these respondents, 66% said they run their applications in containers, such as in the microservices architecture, which is in some way used by 40% of the respondents. These trends encourage the use of orchestration management tools, and Kubernetes is fairly popular, being used by 40% of those who have knowledge of infrastructure development. This is a huge slice of those answering, as the next most used technologies are tied at 14%. In Stack Overflow's popularity survey [67] the number of professional developers who use Docker or Kubernetes stood at 39.2% and 12.9% respectively. Underpinning their popularity is the fact that they are "loved" by more than 70% of those already working with them, and out of those who aren't actively using either one, 24.5% and 18.5% respectively expressed a desire to do so in the future. The survey's correlation chart also shows that these two technologies are actually used together in a big portion of cases.

2.2.1 Case Studies

Next to statistical data there are many individual cases where the use of Kubernetes was a conductor of growth and innovation. Companies from many sectors shared their individual use cases where tools from the ecosystem helped solve problems that arose from increased demand or the exponential growth in technological capacities [37][1].

[1]These are excerpts from the range of studies that can be found among the case studies hosted by Kubernetes.

The more obvious advantage is the agile infrastructure that development teams can be provided by setting up a managed or self-servicing cluster system based on Kubernetes or one of its custom implementations. Among others AppDirect, BlackRock, Pearson and Spotify all use a Kubernetes related solution for ensuring quick deployments and the iterative process of implementing small changes without costly time delays. The ease of deployment and continuous integration also means that companies like Booz Allen Hamilton or Babylon can immediately respond to changes in government regulations, laws or professional demands that require fundamental changes in the services provided by them for federal bodies or speed up clinical validation of their AI models, respectively. Implementing such changes is easy, as the services are small, agile components of a whole application and their testing, validation and monitoring is done in a decoupled system that responds well to a moving environment.

Besides advantages in the development process, the possibility of moving away from infrastructure specific implementations helped users like CERN and Nokia in streamlining their processes and speeding up their workflow. The diversity of tasks that Kubernetes can handle shows here, as the former uses clusters to outsource computing tasks to hybrid and public clouds and also builds their distributed data storage using Kubernetes, while the latter implements telecommunication applications that are used in a diverse set of infrastructures, all requiring their own operational tools and methods. Portability is a requirement that companies such as Babylon value, as they aim to provide services in multiple geographical areas, and uniform, centralised access to their resources is a main concern. The benefits are also expressed by numbers. Deployment times drop from days or hours to a matter or minutes, the number of services built is exponentially higher after integrating orchestration and the costs of maintenance, operation or migration are lowered as well all across the companies that shared their experiences. The scaling ability of Kubernetes is an outstanding point of benefit in most of the cases and the community surrounding the tool and its ecosystem actually drives companies to implement the technology and actively contribute to its success.

From these statements we can deduce the following general attributes users find positive in Kubernetes: agile and easily maintainable architecture, relatively quick setup and streamlined operation, a high degree of abstraction over the system in place for different sections of the teams working with it, high availability and scalability, and the polymorphism it shows when applied to a diverse set of problems. We can also see that while it has ways to go, Kubernetes and the technology most often associated with it, Docker, will be defining parts of the technological landscape for the years to come. With the solutions they offer for highly available, scalable and agile microservice-based architectures, these tools will be the basis for the evolution of modern software design and architectures.

2.3 Architecture

The Kubernetes architecture is a sprawling system of specialised and interconnected components. A deep dive into it would be out of scope of this thesis. This section will provide an overview of the top layer and describe the components that are needed for a fairly basic but serviceable setup based on the work that serves as basis for this writing[2].

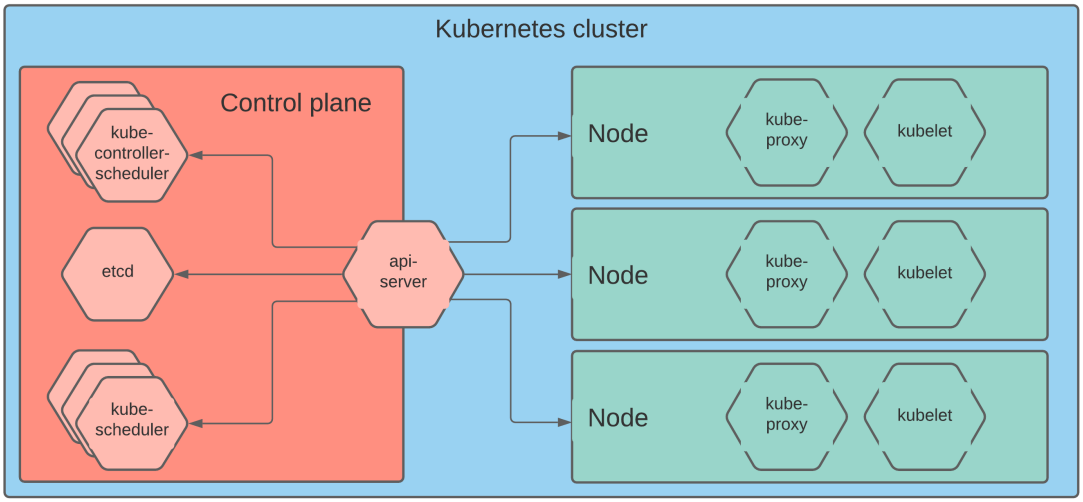

Figure 2.1: A basic Kubernetes cluster infrastructure.

2.3.1 Cluster Infrastructure

When a Kubernetes infrastructure is deployed, it is called a cluster. Figure 2.1 shows the basic components of such a cluster. In this cluster two things work on the top level: the control plane and nodes. Nodes are the working elements in the cluster, they provide the resources and the environment for objects that enable services to be run. Nodes can be run on physical or virtual machines. Their three components clearly describe their responsibilities:

- kubelet: an agent that makes sure that containers are up and running in a Node, i.e. the manager of container life inside a Node.
- Container runtime: a runtime client that is used for operating the containers that are to be run on a Node. Possible third-party runtimes are containerd, CRI-O and Docker CE on Linux [38][3].
- kube-proxy: a proxy that provides networking services for a Node.

By deduction from its working parts, a Node is the general management item that can be used to bundle services run in containers and their necessary resources together and to manage its life, monitor its operation, instrument changes and provide connectivity.

These Nodes are overseen by the control plane. There is one control plane with tools handling management tasks and there can be multiple Nodes under its supervision. It is responsible for instrumenting the Nodes' environments and managing their operation, reacting to changes in their states and scheduling. There are many in-built components, only the most important are detailed here.

The control plane is separate from the other parts of the cluster and communicates through the api-server component with Node instances as well as end users, while also providing

[2]The source of this section is the Kubernetes documentation [40]. The described version is v1.20, but most of these concepts are generally applicable.

[3]At the time of writing a deprecation notice was issued for Docker CE after Kubernetes version 1.20. This is a move towards favouring clients that use interfaces natively supported inside Kubernetes and a general shift towards full compatibility [48].

a way for components to interact with each other. Standardised function calls make it possible to concisely manage and query Nodes. The kube-scheduler provides scheduling functions for Pods that need Nodes to run on. It conforms to different requirements in performance and resource provision and schedules the lifecycle of these Pods accordingly using available Nodes. These nodes inside the cluster require controller processes to effectively carry out intended changes and react to manipulation or failure. Generally, these procedures aim to converge intended and current states; however, special controllers are used for specific purposes such as endpoint connections or failure recovery. These are handled by the kube-controller-manager. Finally, a persistent storage solution, such as etcd acts as storage for data about resources inside the cluster.

The aforementioned elements make up the most basic cluster infrastructure that is able to serve users; however, for different use cases, supporting services and objects have to be used. Nevertheless, these components are always present.

An important point is how an operator can interact with this infrastructure. As mentioned, the API server provides connectivity for the control plane. The API design is defined according to the OpenAPI specification and on the most basic level server functions can be used with REST calls; however, many command-line tools are available and client libraries make programmatic management possible. One of these CLI tools is kubectl [42]. It allows command-line interfacing with Kubernetes clusters for carrying out all operations tasks that the system provides. API objects represent the resources inside the cluster infrastructure and changes are effected through their manipulation. While these are able to be interacted with individually, it is easier to provide the YAML declarations mentioned before, and let the control plane handle convergence and management.

2.3.2 Kubernetes API Objects

Kubernetes API objects are used to describe the configuration of resources inside clusters. These can be provided with YAML files through the API server. These files contain a specification section that is used to describe the desired state by specifying the attributes the object needs to have, e.g. a Deployment spec has a description of exactly how many services it needs to oversee, these services can be configured as a set of similar Pods and the number of Pods to be run can be given. In summary, API objects represent the resources handled by Kubernetes. More precisely, "a Kubernetes object is a 'record of intent'–once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's desired state"[4] [41] One thing to note is that in some cases these objects do not represent actual objects to be created, they can also represent settings such as routing settings in Service or Ingress controller objects, detailed later in this section.

The most basic object is the Pod. Pods are environments that house one or several containers. Pods provide storage and networking resources, such as storage volumes and a single shared IP address for these containers. The scheduling and context handling inside a Pod is shared across everything that might run in it, such as a host would be in a networking environment. This means that containers running inside them can interact with each other without the need for routing between them because they share the networking namespace. Additionally, they must coordinate how they interact with outside objects, as they also share networking resources such as usable ports. An important feature to

[4]This section only aims to introduce object types in a basic manner as needed to understand this thesis' underlying cloud infrastructure.

remember for any operator is that these Pods are discarded if changes have to be made. On reconfiguration it's not the old Pod that is being recalibrated but a new Pod being set in its place with the new configuration, assigned a new IP address and if not provided persistent storage volumes, new storage. This brings with itself the need for care in designing the applications and challenges with the dynamic handling of Pod lifecycles.

Pods are one-off objects by themselves, they can be set to automatically restart upon failure; however, they are scheduled only once to a specific Node. This means that once allocated to a Node, it will run as long as it does not suffer failure and is terminated or its underlying resources are not taken away, like when the Node loses the computing resources needed to maintain the Pod. They are not recreated automatically in this case, and even after recreation, the Pod might be similarly configured but it will be a different Pod. This means the operator has to supervise them and handle problems manually in most cases. Other than handling Pods individually, ReplicaSets and Deployments can be used. A ReplicaSet contains a Pod template declaring the specifications that its Pods must have and the number of Pods it must maintain. It ensures the existence of a set amount of similarly configured Pods it identifies by preconfigured selectors. While this object provides an abstraction over individual Pods, it is easier to maintain a set of Pods with Deployments. They offer additional functionality for managing and reconfiguring sets of Pods. A Deployment maintains ReplicaSets. It contains the same Pod templates, the number that must be maintained of those and a selector for figuring out which Pods it must manage. It is also capable of rolling out the deployment of these in a controlled manner, managing a steady initialisation instead of pulling up all at once. The rollout history is kept so that previous rollouts can be reinstated in case of problems. If the Pod template is modified, a new ReplicaSet takes the place of the old one. The graceful handling of the movement of these Pods from the old ReplicaSet to the new one is possible in the same manner. This works by scaling down the Pods of the old one and steadily filling up the spot with Pods from the new ReplicaSet. These objects make dynamic failover of services possible. By declaring the template (number of Pods, intended specification, file images, etc.) they have a source of intended state which they can follow and handle failures effectively. A ReplicaSet of n Pods will always have n Pods, either in an active state or under initialisation due to a formerly active Pod failing or being reconfigured and replaced.

In Listing 1 an example Deployment of Nginx instances can be seen from the Kubernetes documentation [39]. In essence, the YAML file stores key-value pairs, so stored attributes can be referenced by their dotted notation. Data inside metadata contains information on the API object. This declaration makes a Deployment called "nginx-deployment" (metadata.name). Data inside spec contains configurations the Deployment must use. The spec.replicas value instructs the managed ReplicaSet to maintain three instances of Pods. The spec.selector.matchLabels value (here it is "app: nginx") contains key-value pairs which each Pod must contain for them to be connected to the Deployment object, i.e. establishes ownership by the specified Deployment. The spec.template declares the template specification of the Pods that are supervised by the Deployment, the intended state at which they must be. Inside this, the metadata serves the same purpose as before, with the addition that labels must contain the selector pair mentioned. The template.spec contains the specifications for the Pods to be run such as its name, the name and tag of the software image and the ports that must be open.

This dynamic process, however, makes it hard to implement a static IP addressing scheme, as a new Pod is given a new IP address on initialisation. A Service object solves this problem and also provides load balancing. A Service object can be specified with different

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Listing 1: Example Deployment of Nginx containers.

selectors which can point to metadata in Pods and the object will provide stable network addressing to and load balancing between them. This object will have its own IP address as well, but it can also have a DNS entry. If DNS functionality is present inside the cluster, Pods that require other Pods not behind their own Service can use the Service's DNS name to query its IP address. After connecting to it, the Service then routes to the Pods behind it, establishing connectivity between Pods behind different Services. It does not have to change with the Pods that it connects to. This way a set of Pods with attribute "x" configured for them can change in whatever way, as the Service that is set to discover Pods that have "x" configured in their metadata will provide routing to those Pods at any given time. From the outside in relation to this Service, its IP address x.x.x.x can always be used to reach the Pods behind it, as no matter the permutation of the Pods' IP addresses, the static Service handles discovery and routing. This routing will happen in a balanced manner, all Pods will receive a share of the network traffic.

In Listing 2 an example Service has been declared for the previous Deployment. As before, metadata contains information, spec contains configurations. The metadata.name value will be the DNS entry that can be queried to reach this Service and thus the Pods covered by it. The spec.selector has the key-value pair which is used to identify Pods that this Service must cover. In spec.ports a list of key-value pairs define firstly "port", which is the port the Service opens to the outside, and secondly "targetPort", which is the port that traffic from "port" will be routed to. In this example these are numbered; however, it is advisable to name the referenced ports and use those. Multiple of these pairings can be made, opening up more ports. The used transport protocol can be defined as well.

The aforementioned are the basic fabric that enable the cluster to provide dynamically scaling and highly-available services, illustrated in Figure 2.2. These cluster resources

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

Listing 2: Example Service for Nginx Deployment.

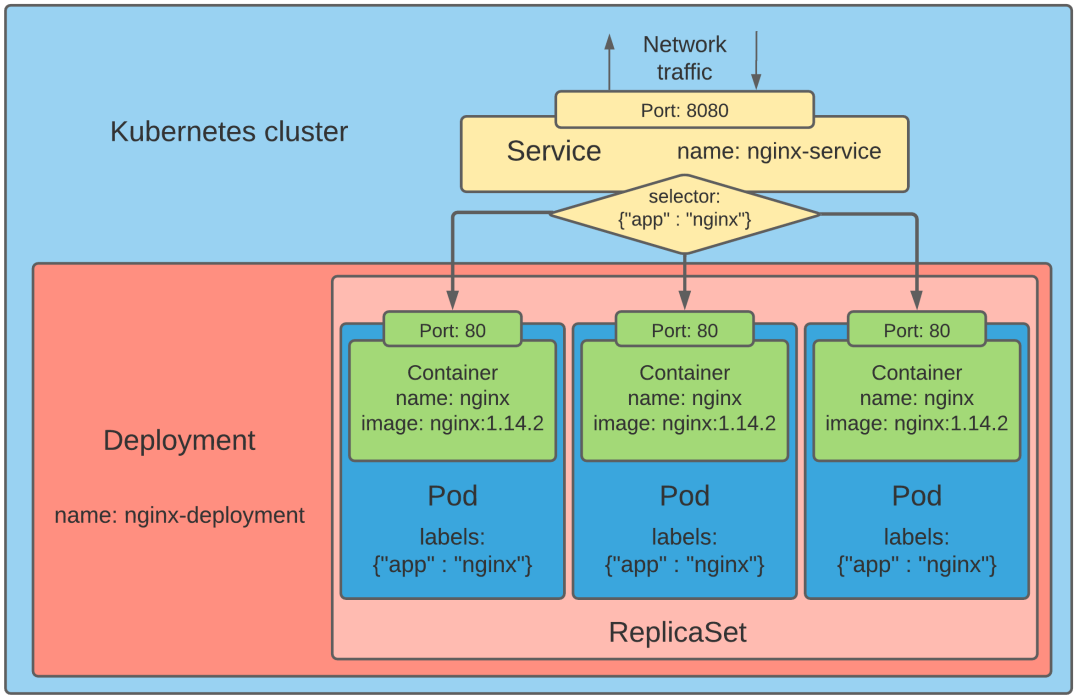

Figure 2.2: Visualisation of the cluster setup defined in Listings 1 and 2.

handle the lifecycle of containerised applications, the management of their resources and communication between distributed components. Additionally, Kubernetes allows for custom resources to be included inside a cluster setup. These objects are third-party software that need to provide their operation workflow through Operator API objects that describe the maintenance and operation of the software they were written for. The following two are examples that have been used inside this thesis' cluster environment for additional functionality.

- An Ingress Controller is an object that provides connectivity between the public internet and the inner network of the cluster. It is the way of opening up services through an ingress resource like nginx or other provider specific technology. Routing can be configured so that different paths open different services. By default, HTTPS is required, so a method of certificate handling has to be in place.
- A Certificate and a Certificate Manager object are resources enable dynamic management of certificates. The former describes the domain attributes that have to be registered and the latter can handle the automatic requisition of certificates with preset Certificate Authorities and certificate attributes, e.g. test certificates for production testing.

On top of the mentioned elements in this section, Kubernetes provides a lot for operators and developers alike to make cloud-native systems and services. It can be configured and implemented in different ways conforming to different workflows and environments. It is a tool that can propel information technology towards cloud-native operation.

Chapter 3

Software Design Techniques and Technologies

A main part of this thesis' practical work is to carry out the development of a simple software, so that it can be used in other parts of this thesis. The aim of this software, housed in a cloud infrastructure, is twofold:

- adhere to the best practices of cloud-native development and design so that the application would showcase the advantages of the approach while operating such software in a cloud environment and
- implement metrics exposition that would make metric collection and monitoring possible, and log generation while conforming to the scalable and dynamic nature of cloud-native operation.

In Sections 3.1 to 3.3 the design patterns will be detailed that were used in the development of this thesis' application. As stated, it was an expressed goal that this software would be implemented in a way that would conform to cloud-native principles. Modern computing solutions enabled new methods of service delivery and these novel practices presented questions that had to be solved in a different way. The way data is stored, accessed and kept consistent, the decoupling of business logic and the optimisation of process handling are some of these.

Questions about the efficient way of developing and deploying services as a team effort were also raised. The guidelines that must govern the development of applications that are to be used entirely in a cloud-based environment and managed with tools that handle the lifecycle of such applications dynamically have been aggregated over years as commonly accepted principles. These observations and recommendations have been made with the experience that comes from the increasing growth and implementation of the cloud infrastructure. Section 3.4 will cover the general principles, those that emphasise a framework which should be followed by any project aiming to be cloud-native, so that they can utilise all the advantages of such environments while avoiding common problems and mistakes.

Finally, in Section 3.5 a summary of all principles detailed in Chapter 3 will be aggregated into a framework that defines what cloud-native development means in relation to the software development work carried out in this thesis. This thesis does not have the scope for aggregating and comparing all the principles and patterns, this description will aim to describe only those that have been chosen and utilised.

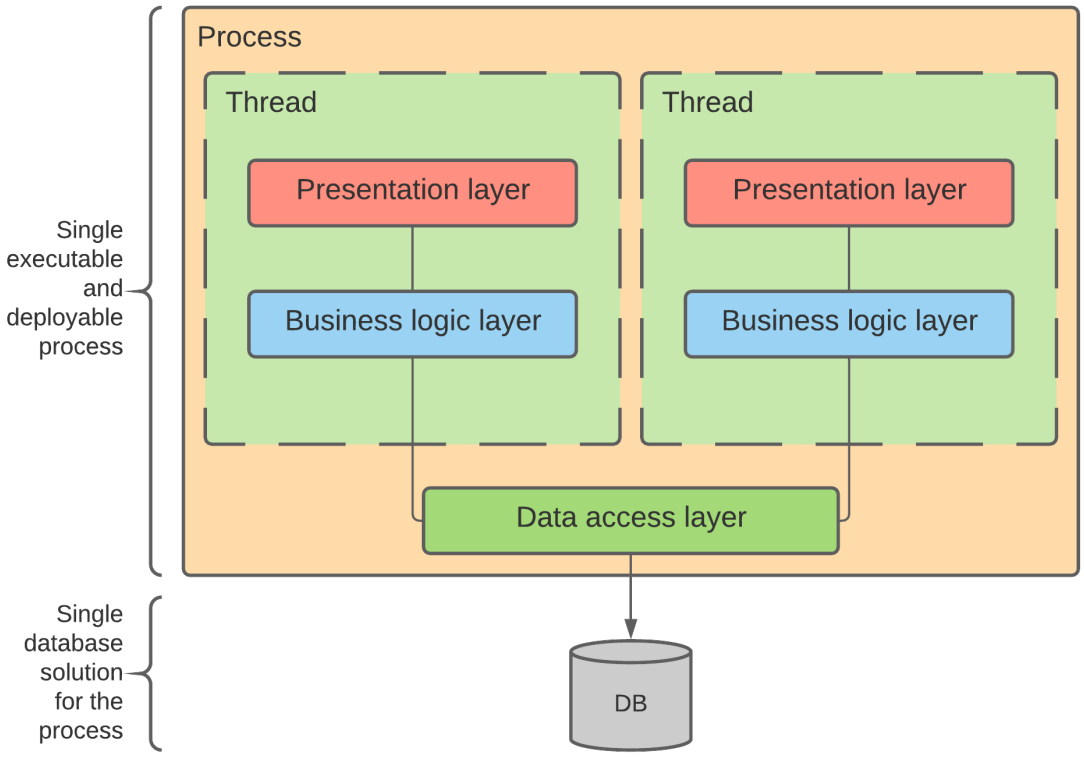

Figure 3.1: An example layout of the monolithic architecture.

3.1 Microservices Architecture

The microservices architecture aims to streamline the workflow of modern software development teams, which are made of small, agile groups, responsible for a single product throughout its whole lifecycle. That cycle is self-repeating, new problems spur on development towards new directions or deeper in the methods these teams already employ. This agility and the ability to quickly append or modify, and then release their product would be hard to achieve if their products were interdependent on other teams and products.

This approach to software development contrasts the monolithic approach, exemplified in Figure 3.1. This covers software built as a single executable that can provide multiple functionalities, i.e. multiple services. These software are run as a single process, with multiple threads where needed. A characteristic of this approach is that any time a change is made to a function included in the software, the whole executable has to be rebuilt, tested, and released. The development of each individual function has to take the development of other included functions in regard, which causes delays or incompatibilities when changes need to be introduced at a moment's notice. The monolith can be developed with agile methodologies; however, using microservices to build an ecosystem of independent services cooperating with each other greatly reduces stress on the teams working on their applications and comfortably aligns with cloud-native development and operation.

Martin Fowler and James Lewis have discussed microservices in detail on Fowler's blog [58] and the trio of Michael Hofmann, Erin Schnabel and Katherine Stanley [61] has written many observations about the best practices to be followed by teams working on

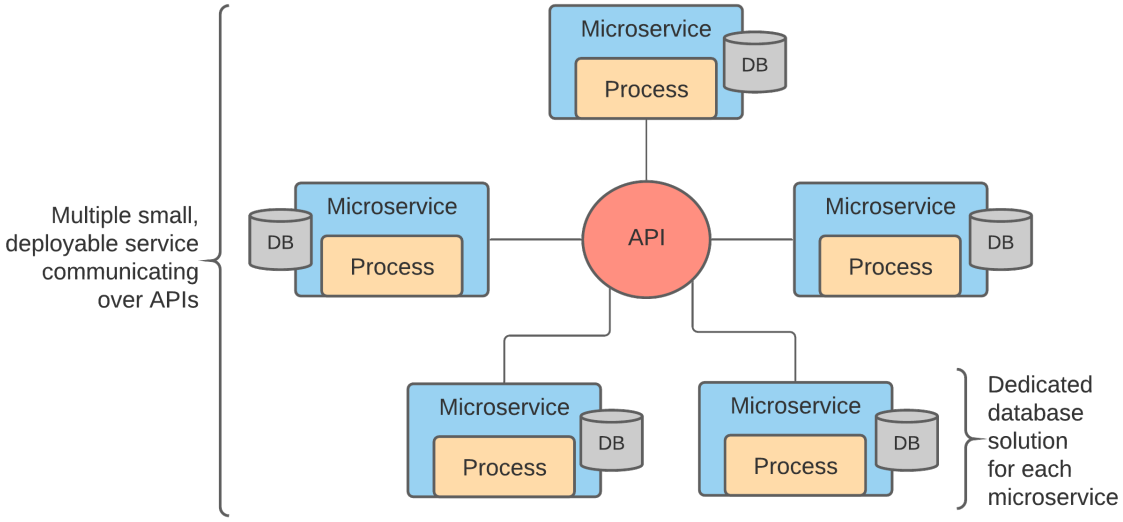

Figure 3.2: An example layout of the microservices architecture.

products employing this architecture. Their works provide the basis for this compilation of characteristics and recommended procedure[1].

3.1.1 What Are Microservices

Based on the summary by Fowler and Lewis, microservices are standalone services running their own processes aimed at performing a well-rounded functionality. They communicate through lightweight APIs with HTTP and are able to be deployed independently with minimal central supervision. This suite of interconnected services is the microservices architecture. Hofmann et. al. break this down concentrating on "micro". The word micro in microservices does not denote the size of the codebase but rather the purpose of the service. "Microservices should do one thing, and do that one thing well" [61, Page 4]. This means that during the design process the purpose of the application should be decomposed into smaller, interconnected purposes, up to the point where small, well-defined, autonomous and independent services make up the application's ecosystem, such as the one illustrated in Figure 3.2. In the simplest terms, microservices are a way to compartmentalise the development of large applications by delegating the development of well-defined tasks to small teams.

3.1.2 Independence and Autonomy

For Fowler and Lewis, the monolith has proved difficult to continually develop, as even though changes would have to be made only to a component of the application, the whole structure had to be released and integrated after any kind of changes. Scaling also requires considerably more resources. A monolithic application is better scaled vertically by increasing resources of an already running instance; although, this is costly. The microservices architecture has been drawn up to solve these problems. The microservice approach enables interdisciplinary teams to work together on defined business capabilities. When teams work on their own architectural layers, cooperation between teams for the

[1]While not directly influencing this work, Chris Richardson's blog [69] and the blog produced by F5 related to Nginx [21] are main sources of guidelines relating to microservices patterns

optimal handling of implementation challenges is hard, as all decisions made by the team has to be relayed to other teams, taking time away from development. However, with small, cross-functional teams individual services can always be developed in the most optimal manner. With dedicated specialists, all architectural layers can be built according to the needs of the well-defined business problem at hand. The fact that the technologies used are not centralised also encourages ownership of product by their development teams, they can make the best decisions in the development lifecycle of their service.

As of Hofmann et.al, autonomy and independence are important features of microservices, as changes implemented can have widespread influence if not properly bounded inside individual services. These microservices should encapsulate implementation details and data structures, data sources must be private to each. This enables the developers to frequently refactor the service as inevitable changes have to be made, which is a strength of the architecture due to its independent structure, not a setback. Such a decomposition of services makes way for individualised implementation. This in turn enables teams to introduce solutions that best conform to their private requirements. The fact that from logic to data representation all aspects are in the hands of the developers enables the system to be polyglot. Polyglot means that all services can use the exact language and data service that they think best for their purposes; however, the system must operate on the basis that not one service knows the implementation details of another when communicating with it. Nevertheless, on the design side, it is advisable that the code base of the services share more than standard libraries, if possible, so that the functions can be implemented as independently and concise as possible. If a set of specialised libraries is used in at least a few places for example, the functions provided by those should be examined and possibly moved into its own microservice.

3.1.3 Communication

Fowler and Lewis define the frame of communication methods to be used in this architecture. Microservices use simple communication protocols such as HTTP, so that there is minimal logic in the method of transmission itself. The business logic is based in the loosely coupled and cohesive microservices. Even there, the basic function is receiving a request, applying logic and producing a response. On the practical side as of Hofmann et.al., illustrated in Figure 3.3, the communication methods can be synchronous, RESTful protocols or asynchronous message queue protocols, and it is always best if they are language-agnostic. The same can be said of data formats with JSON being a favourite. The decision between synchronous and asynchronous communication methods should be made considering the exact requirements of the system. The specifics should be clearly documented and leniency should be exercised in handling payloads and headers as new consumers could always join, which are not up to date on the current state of communications.

The services use APIs to facilitate this communication. When a service needs to interact with another service, it uses the service invocation method provided by its environment either on the server or client side, such as an API gateway or sidecar processes running next to the service itself. The amount of preconceived functionality from these should be minimal, while the handling of errors should be as holistic as possible. The microservices architecture places great emphasis on fault tolerance, as a simple, unexpected fault in one implementation could bring down the whole stack if not handled gracefully. Because the system is distributed, errors are encountered in communications between microservices or by internal exceptions, only the former is related exclusively to the architecture. The

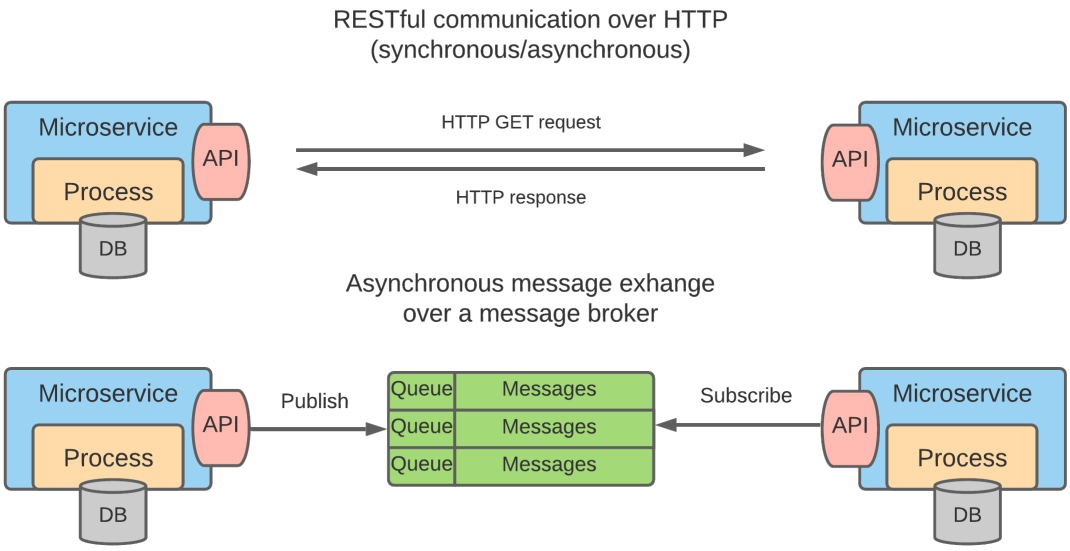

Figure 3.3: Two of the possible communication patterns in the microservices architecture.

propagation of errors should also be avoided, errors have to be handled locally so as to not introduce additional complexity with regards to system-wide fault tolerance. The best practice is to assume that bad requests and errors will always happen, so as to preemptively handle bad use cases. Asynchronous communications is generally recommended, except in cases where acknowledgement is necessary, thus event-based protocols are a good choice for distributed systems. Fault tolerance depends on the implementation of services, as they are the ones handling faulty communications. The manner of failure handling should depend on the communications method used; however, generally these should ensure that changing an API does not cause breaking errors inside the system.

3.1.4 Data Management

The distributed management of data introduces challenges with updating the data stored. As each microservice stores its data privately, distributed transactions would have to be made in order to consistently update data; however, this is very hard to implement. Fowler and Lewis propose transactionless coordination of data updates that might span more than one microservice, where interactions and compensating operations handle changes in data stored or errors encountered. This way a degree of eventual consistency is present inside the architecture, but the benefits of agility can outweigh the costs of momentarily inconsistent data. Hofmann et.al also conclude, that each microservice should handle its own data privately, thus it should have its own data store. This enables polyglot persistence, the state where data is persisted in a multilingual environment. Every data is stored in a way that best enables its management, with a wide variety of choice between SQL and NoSQL databases, as shown in Figure 3.4.

In this architecture, because of its distributed and heavily specialised nature, data transactions can span multiple services as mentioned by Fowler and Lewis as well. However, joining data from multiple sources is hard and consistency has to be ensured. This can be handled in multiple ways: adapter services handling such transactions, introducing an event-driven architecture with publish/subscribe communication pattern or merging ser-

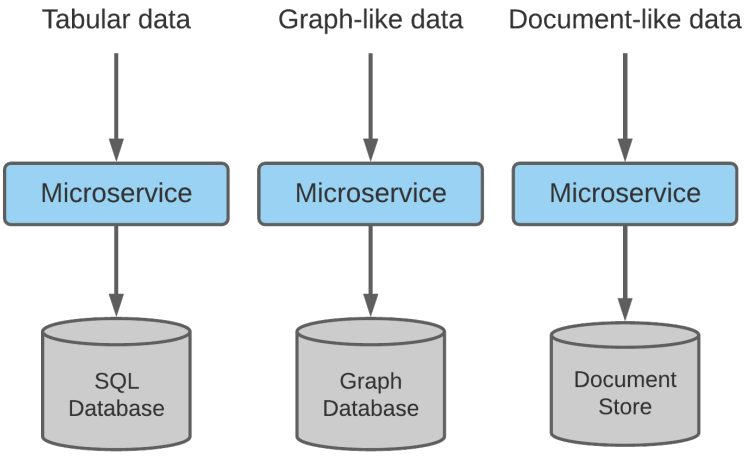

Figure 3.4: Individual data handling by microservices.

vices based on the frequency of event exchange. An example for an event-driven solution is event sourcing and Command Query Responsibility Segregation, described in Section 3.2. Whatever the solution, data stores and services must remain decoupled and data consistency ensured[2]. The CAP theorem[3] provides a point of reference for the effect of design choices. Teams have to consider the costs and benefits of either the necessity of handling eventual consistency or being unable to provide the best availability.

3.1.5 Handling Failure

Fowler and Lewis concludes that the build of the architecture also means that developers have to prepare for one or more of the microservices failing. Real-time monitoring is an important part of this system, as problems have to be noticed and acted upon by the development team. The failure of a service can seriously impact the users' experience with the application. Because of the size of these services and their independence, scaling them can solve problems stemming from overload. As of Hofmann et. al., microservices are designed to be easily scaled horizontally by running new instances rather than increasing provisioned resources. Dynamic provisioning is the ability of a microservices-based system to scale applications according to user load and make sure that resources are always optimally utilised, i.e. reserved for only the necessary amount of running applications. Based on provision regulations and momentary utilisation, the system scales the system accordingly. This also ensures that unhealthy or dead instances are dismantled, so that they do not use system resources. This is made possible with the use of health checks, functions inside the applications or their runtime environment that indicate functionality. When operating a microservices architecture, it is important to be able to ensure communication between services, to scale each service according to momentary load and supervise the health of the services, ensuring that failover happens. This is done independent of the service's running process and introduces the problem of other parts of the system discovering these new services, but also the possibility of dynamically load balancing between

[2]For more detail on the possible structures of data persistence see Hofmann et. al. summation [61, Chapter 5] or Chris Richardson's collection of patterns for data management [69].

[3]Formally introduced by Eric Brewer, it states that because tolerance of network partition is a necessity due network errors, a trade-off must be made between the availability and consistency of the data stored inside a system. For an elaborate discussion see his article on InfoQ [47].

them. A service registry solves these, and the following aspects have to be provided either by the register or the microservice when using this method:

- registration and deregistration of services,
- heartbeat signals that indicate a healthy service
- and service discovery for components of the system.

The system changes dynamically and under heavy, fluctuating load, frequently. Logging and the use of metrics is fundamental in this architecture, as well as any other system, for noticing problems in a timely manner and collecting observations for further development. However, the fact that many services might interact makes traceable logging of interservice events indispensable. Singular log entries based on an individual system only helps in figuring out errors with the service itself. When multiple services' interaction causes an error, one service's response to a faulty action might not tell how much influence the action of another service has on its own operation. Troubleshooting problems that span multiple services is made easier if connected events are bounded, such as by an identifier. A log collection solution also should be integrated so that these logs are easily aggregated and observed. If data visualisation tools are used, having a single source can make reporting easier. The dynamic lifecycle of both the services and the collector should be taken in regard and connecting dynamically changing service instances to a log collector has to be solved.

3.1.6 Summary

Such a complex system puts a lot of operative strain on its developers and operators, it is proposed by Hofmann et. al. that a good level of automation has to be introduced. With so many moving parts, understanding and keeping the structure in mind is hard, so checks introduced in automated integration and delivery systems help keep the work of independent development teams streamlined. Detailed discussion of ways and methodologies for automation can be found in Section 3.4.

The microservices architecture is a response to frustrations by developers regarding the increasing speed at which they must develop monolithic software. The possibilities provided by cloud computing also means that operating large structures as single units is not always the best method. Martin Fowler does not purport this architecture to be the solution, rather a way for software development to evolve. In his cited writing he handily provides insights into the effect his mentioned methods can have while integrated into a monolithic development structure. Nevertheless, microservices are popular. According to JetBrains' 2020 survey, around 40% of respondents work with microservices in some way [64]. As summary, Hofmann et. al. list four general guidelines that should drive the design of microservices, so that they are best placed to carry out their purpose with optimal performance [61, Page 89]:

- A microservice must be independently deployable.
- A new version of a microservice should be deployable in minutes rather than in hours.
- A microservice must be fault tolerant, and should prevent cascading failures.

- A microservice should not require any code changes as it is deployed across target environments.

If these points can be fulfilled, the system maintained by the development team is ready for the agile methodologies employed by modern software development teams to quickly release and maintain software solutions.

3.2 Event Sourcing

Event sourcing is an architectural pattern which was first introduced by Greg Young in 2007 as expressed by Young himself in a conference talk describing the pattern [77]. Martin Fowler[4] [57], Chris Richardson[5] [69] and Young have elaborated on it over the years, finding ways to make event sourcing a popular method of designing large-scale systems with a heavy emphasis on auditability, the ability to separate effects from their originating events or retrospection. This summary of the design pattern is a general overview of different viewpoints based on their works. Descriptions of the aspects of domain-driven design are based on Eric Evans in his conference speech at Domain-Driven Design Europe 2019 [52].

The whole architecture is based on elements from event-driven and domain-driven design; however, it can be standalone and not dependent on them. Data is represented by interconnected events stored inside streams and state can be derived from the sequence of these events, illustrated in Figure 3.6. According to Young, all businesses that have matured to a point where they are strictly liable for data they handle are naturally event sourced, and as such, the accountant's example is used by him frequently. Accounting does not handle accounts such as bank balances as states updated inside a ledger but rather as a summation of previous transactions that have led up to the point that is being investigated. If there is a discrepancy between what the accountant's customer believes and what the accounting shows, not having a backlog means there can be no way of proving the truthfulness of the current state. However, by storing the events that lead up to the purported state, a common source of truth can be established and every state will have a descriptive chain of linked events that serve as a basis for the claims of the side liable for handling the data. Event Sourcing puts this into practical terms in software design.

3.2.1 Practical Design of Event Sourced Systems

An architecture based on this design has an event store and services that utilise data inside this store. This event store has to have two capabilities: sequenced storage of data and implementation of the subscription pattern. This means that this store can be the single source of truth inside this system as it has the ability to persist and distribute every event that happens. While the services surrounding this store can have their own database (and they mostly do in memory at the minimum), any state is in a way a derivative of accessible events provided by streams inside the event store. This basic structure is illustrated in Figure 3.5.

The sequential read of the stream can be implemented in different ways, and APIs to different message broker and message queue solutions provide methods for using this technology,

[4]Contributed to the codification of the design with his work, such as his personal blog.

[5]A proponent of the microservice design principle with a separate place for event sourcing on his personal blog.

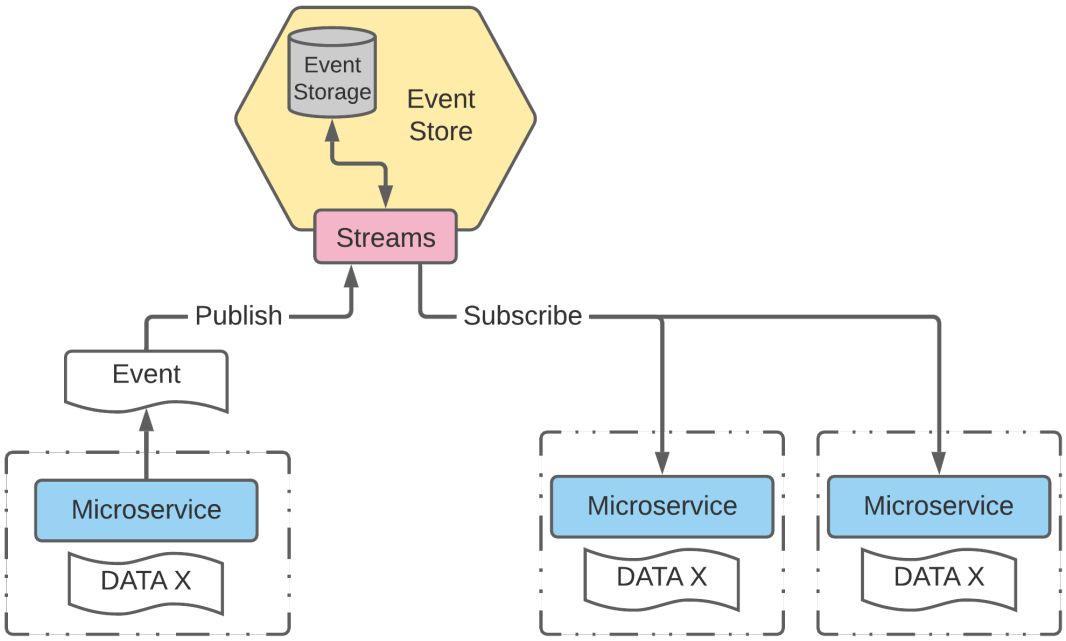

Figure 3.5: Example microservices exchanging data using an event store and the publish/subscribe pattern.

such as RabbitMQ, Kafka Streams or Redis Streams. The technological background of these are different because they were designed to reach different performance goals, these APIs were built on top of an already established solution. Care should be taken when choosing one, as while it is true that all are able to carry out tasks required of an event store, they fulfil these differently, especially in this aspect the manner in which they distribute and persist data. Greg Young is part of a project that aims to bring together all positive aspects of these and similar solutions and provide the exact tools that make an event store effective, the EventStoreDB[6]. Some of the attributes that make it work out of the box for event sourcing projects according to their website: the in-built versioning of events and the possibility of using expected version arguments in calls to the store to ensure consistency, the various types of subscription models provided and the idempotence of applying events to a stream, in effect ensuring that there is no problem with duplicate events, a possible problem in distributed systems.

The services around this store implement the domain logic that interpret the data that is stored there. This approach easily overlaps with the microservices architecture. For most, only the domain logic has to be distributed in these services, and even then, only the ability to interpret this logic is necessary for all the services to have. If a service only consumes events, there is no need for backwards compatibility, as it only reads the state of the system. Where this is not the case, however, special care has to be taken so that the events emitted by the service conforms to the logic that the system operates by. This is one of the pattern's main strength and a disadvantage as well, as this can prove to be a steep learning curve, while providing ways for already up to speed developers to implement domain logic not easily described in traditional, column-row data stores such as SQL-based databases. Where the latter might require layers of abstraction between the

[6]The project can be reached on their official website [35] and the code can also be browsed on GitHub [34].

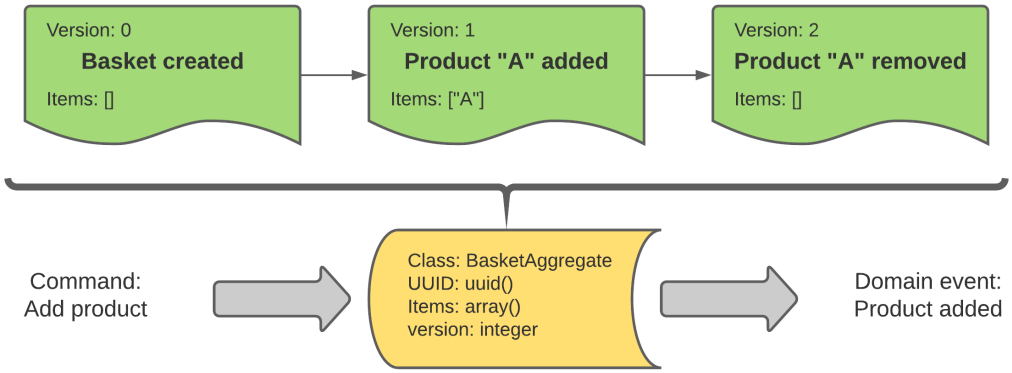

Figure 3.6: A sample event stream with example for the Command - Event structure.

implementer and the domain representative[7] in data representation schemes, the former can bring them together, and by using a shared domain language[8], make the iterative process of development faster and easier, as the representation is based on this common language.

Martin Fowler mentions[9] [59, Part 1, Chapter 2; Part 2, Chapter 9] two ways to implement the connection of events and business logic: the transaction scripts and the domain model[10]. The two are fundamentally different; however, they share the same purpose. The service that uses them interprets a user intent and translates it into an event. That event is sent to the event store and distributed to all interested services where they are interpreted and acted upon. This in turn can cause other events, causing their own consequences. This makes it easy to abstract away the method of implementing such logic, employing the grammatical structure from domain-driven design: there should be a command (a verb) and an event (a noun). The command can be tied to user intent[11], essentially covering the actions that the user can perform via the service. At this stage, the intent goes through the business logic, encountering all the logical checks implemented. If the intent is actionable, the event is formed and emitted, thus making the intent into reality inside the system. This is a good overview of how services and the event store can interact.

No matter the approach, the state should be stored inside the service. A logical representation that is updated according to the events that the service subscribes to allows local modelling of the system's state. The above mentioned two logical implementations are tied to this aspect, state can be stored by objects or other types of structured data. This way the logical checks that allow or deny an action can be executed on readily available data, without the need for constant read requests to the event store. Two problems are raised by this, however. Firstly, the state represented by a service in a small time period

[7]Domain representative is the person with knowledge in the field the implementer services. Examples are an accountant, a logistics professional or a stock broker and the developer working on solutions for them. Software solution patterns might not match domain patterns important for a lot of fields.

[8]An aspect of domain-driven design, where the domain representative and the software developer make and share a common language that has to guide the development of the services provided.

[9]A short writing accessible on Fowler's blog. [54]

[10]The former contains the logic in scripts that are executed based on the type of transaction, the latter contains objects that represent individuals that are meaningful to the system such as a whole institution or a single order form.

[11]Intent has no additional meaning here or connection to intent-based patterns, it is only a turn of phrase.

might not actually conform to the state that the system is actually at, due to another service instrumenting changes via their own events. The ripple effect would eventually reach this service and converge its system model to the actual state. Nevertheless, information supplied by this service is inconsistent for a period and eventual consistency is part of the architecture. Secondly, the business logic operating inside this service might act on data that is not in a consistent state and this should be carefully mitigated. Versioning helps in this case. If an action is to be initiated, version checks should always be a part of the transaction. If the version of the stream that contains data the action intends to append is different from what is kept by the service, error handling processes should be implemented based on the kind of difference between the service's state and the actual system state. If the system state is newer than the service's state, the actualisation of the service through reading the stream and rerunning or denying the action can help. If it is the opposite way, the possibility of faulty operation should rise and the discarding of unregistered events should be handled. After that, the service's state should be re-synchronised and the action should be rerun or denied.

In the previous paragraph, the problem with too many read requests was raised, and blocking the single source of consistent data counteracts the advantages of the pattern. Additionally, since reading data is reliant on the logical evaluation of sequential events, querying the state of the database is difficult. So, if frequent read action is required to carry out services inside the system, the separation of read and write models can help. The Command Query Responsibility Segregation (CQRS) is a pattern that works in tandem with event sourcing. It stipulates that the system has to include a read-only model responsible for providing data that represents the system's state according to the changes appended to the write-only model. This read-only model separates the concerns of servicing read requests and transacting write requests, and also allows for different types of view representations without changing the write model. However, implementation of this pattern also has to consider eventual consistency [70].

The aforementioned can all be a part of the event sourcing pattern; however, they are borrowed from the patterns that serve as a basis for it, the domain- and event-driven design patterns. Event sourcing provides other benefits that set it apart from the rest:

- The current state can be discarded anytime and rebuilt from the event store, reaching the same state as before. It is also possible to create snapshots in past time by configuring the moment up until which the state needs to be observed.
- If there is a mistake in the sequence of events, there is no need for corrective transactions as the state can be rebuilt up to the point of mistake and taken from there. This does not only mean that an incorrect event can be fixed, but also that events in the incorrect order can be replayed to build the correct state.
- Data stores based on event sourcing can provide a base for auditing the data kept, as it stores the stages data goes through rather than the actual state. It provides insight into cause and effect sequences for root-cause inquiries, enables retrospective analysis of possible outcomes.
- The store provides an observable source of data, where information flows in a single direction with clearly defined responsibilities.
- The fact that events are versioned means that a service can start, restart or reconnect anytime and ensure that information stored by it is up to date by synchronising itself with the event store.

Event sourcing is a performant pattern for fields that attribute increased importance to the manner in which data changes. Combined with the advantages that the connected design patterns bring, systems based on it can provide added value to the consumer, as their domain knowledge can be implemented in more depth. This in turn makes the data stored and the sequence of the events much more meaningful to non-developer representatives. It eases cooperation among developer and consumer as they share a common language. Expected or perceived effects by the consumer is understood by the developer more easily, enabling better implementation of development goals.

3.3 The Twelve-Factor Application

The methodology was described by Adam Wiggins [76] around 2011, then working at the platform-as-a-service provider Heroku, detailing the aspects of software that performs well in a cloud environment. Pooling the company's experience in developing software-as-a-service solutions[12] and hosting others, he posited his principles to raise awareness of the method in which applications for the web should be built. His methodology serves to enable a web application's[13] dynamic growth that comes naturally with the evolution of its services, conform to the dynamics of the workflow that comes with the development of such an application, and avoid the cost of software erosion[14].

3.3.1 Factors

As defined by Wiggins in summary, a web application should be developed in a way that conforms to these points: uses declarative formats for setup automation that enables the quick on-boarding of new contributors, has a clean contract with its underlying system so that it is easily portable and deployable on modern cloud platforms, easily scales up without additional effort, and enables continuous development and production workflows. To achieve this, Wiggins provided twelve points that should be considered:

1. Centralised, versioned codebase
   The code which is running as 'deploys' in development or production environments should share a single codebase in a version control system.

2. Explicitly declare and isolate dependencies
   Dependencies in applications are supporting libraries provided from outside the source code. These should be exactly declared in a manifest beside the application. Dependency isolation means that the application is not influenced by dependencies present in its surrounding system and it only has access to predefined dependencies. Usage of tools by the application should not inherently trust the presence of the tool

[12] The concept, which can be described as X-as-a-Service, is that different layers of operating IT services can be abstracted away and provided as services for customers so that their portfolio can specialise only on a section of such operation. For example, a software-as-a-service provider handles all software related operations tasks for its customers above everything else, while a platform-as-a-service provider offers managed environments which can be used by teams without the need for employing administrators to provision resources or manage operating systems. A Saas solution example is Dropbox or MailChimp, a Paas is AWS Elastic Beanstalk or Heroku.

[13] In this and related sources the terms "web application" and "software-as-a service" are interchangeable.

[14] As quoted by Wiggins: "software erosion is 'slow deterioration of software over time that will eventually lead to it becoming faulty [or] unusable' and, importantly, [...] 'the software does not actually decay, but rather suffers from a lack of being updated with respect to the changing environment in which it resides'" [75].

on the running system. This ensures uniformity across development and production systems.

3. Store configuration in the environment
   Any configuration that might vary between deploys should not be hard coded into the application, there should be a strict separation of code and configuration. The most easily scalable way of this is using granular environment variables for individual deploys.

4. Treat backing services as attached resources
   Backing services are services the application accesses over the network, such as databases or API-accessible services. These should be loosely coupled with the application and no distinction should be made between local and third-party services. Access to these should be defined in the configuration with credentials or URLs. The services behind these locators should be changeable without modification to the source code.

5. Strictly separate build, release and run stages
   An application goes through three stages towards deployment: the build stage converts a defined point in the version history into an executable bundle, the release stage adds the deploy's configuration for immediate execution, and the run stage runs processes of the service. These stages should be separated, as steps in a further stage that correlate to responsibilities of a previous stage cannot be propagated back.

6. Run the application as stateless processes
   Services are executed as processes and any persistent data is stored in backing services. No action taken by a service should be reliant on which process it is owned by. The application should not trust that a system resource, such as cached data, will be available later in the future.

7. Export services via port binding
   The application should be self-contained and not reliant on runtime injection of a web server. As such, it should export its services by binding them to a port and listening on it for requests.

8. Scale out via the process model
   Handling diverse workloads should be based on specific processes. If different types of tasks can be defined, using task-specific processes makes it easy to concurrently run, scale them. For example, an HTTP service and its backing long-running service can take up their own processes. Management of the processes should be left to the operating system.

9. Achieve disposability
   By ensuring fast startup and the graceful handling of predefined or sudden errors, the application should be able to be stopped or started instantly.

10. Ensure development/production parity
    Development teams might use tools better suited to their workflow but different from the production tools. The difference between the skill sets of the development and operations teams could mean that the application will have trouble in its operation and troubleshooting will take considerable effort. To solve this, the development and

production environment[15] should be kept as similar as possible down to the backing services' version numbers. These two teams should also closely cooperate so that every person will have the necessary skills to handle any problem.

11. Treat logs as event streams
    The application should write its logs' event streams to the standard output and not concern itself with persisting it. The handling and storing of these logs are the tasks of the running system environment and these should be transparent to the application.

12. Run management tasks as one-off processes
    If there is an administrative task that must be run, such as database migration or one-time scripts, these should be run in separate processes than the application in an identical environment. These codes should ship with the application source code for synchronisation.

These points can roughly be summarised in three distinct areas: the way developers think about their software design, how the application can be operated in a way that it most seamlessly integrates into a cloud environment, and the cultural aspect of software development. Points number 2, 3, 4, 6, 7, 9 can intimately influence a software developer in the way they approach their task. From how an application runs on a computing instance, through the manner in which they expose their service, to what they expect from and must know about the environment their software will run in, effort has to be taken so that the developed solution will conform to the environment it will be deployed in. Points number 8, 11, 12 give consideration to the way these applications should be optimally run inside any cloud environment and to patterns which can be used to carry out operations tasks such as monitoring, load balancing or administration. The other points, numbers 1, 5, 10 define common grounds for different teams working on ensuring the life of a single service. Having a single source of truth in the presence of a version controlled codebase and adhering to shared practices in building and maintaining similar environments for different purposes serves as a base for what today influences modern software development procedure.

The twelve-factor application is not a distinct application. It concerns many questions that are important when planning for its cloud operation. These factors are language-agnostic, these best practices can be followed by any team in any kind of environment. As is easily discernible, they define a wide array of topics surrounding software development and operation, and are concurrent with how the processes of such methodologies as DevOps, GitOps and cloud-native development have evolved. It does not serve to be a strict framework, and considering the many ways in which software is developed, some points might not be able to be observed as well as intended. However, the most possible adherence to these factors can make the development and operation of a web application the most seamless it can be.

3.3.2 Criticism and Additions

Over time some additional points and criticism were raised. Ben Horowitz detailed the application of the twelve factors to a microservices-based architecture [62] and noted that

[15]The two stages where a software is tested and operated are development and production. The development stage is specialised for the extended testing of software in development with aims for deployment, while the production environment is the actual, operated environment where the services are deployed and accessible by customers.

the original factors were specific to Heroku's own platform. As a whole frame, it is not an exact fit for microservices, popularly utilised in application development[16]. For this reason he provides amendments to the factors that conform better to the solutions provided by him and his company. His additions only extend the twelve factors in a way that fits better with solutions that utilise the microservices architecture, and consider more of the modern solutions that facilitate tasks defined in these principles such as log collection or containerisation[17].

Kevin Hoffman [60] also provided insights into the application of the factors in software development. His most important additions are the three factors he amended the original twelve with: API first, telemetry and security in the sense of authentication and authorisation. Beyond this, he elaborated on the individual factors and provided best practices to each. Also it is a clear notion that he speaks about cloud-native applications in the context of the product that is developed following these principles. It can be deduced, that his intention is for these principles to be considered relevant to applications that are to be developed for environments that are cloud-native in the sense this thesis aims to describe it. The detailed description of specific development and design practices is beyond the scope of this thesis and there are a few that correlate with other practices that this chapter details, therefore the summary of Chapter 3 can serve as a point of reference while in this section, only the additional factors are elaborated.

The API first factor [60, Chapter 2] states that to truly use the advantages of an ecosystem, where individual services consume each others products, starting the development process by defining its API first can lead to better cooperation with other teams and a more streamlined development. By designing the API first and adhering to it throughout the service's development cycle (which is, by its nature, self-repeating), further development and integration into a larger system is easily done. The API can be used to effectively automate the continuous processes in the development and production workflow, it can be used by other teams to integrate their own product with the developed service, and starting from the API design enables the most effective planning of the underlying service to be written.

The telemetry factor [60, Chapter 14] gives further consideration to the facts given in the original factor about log event streams. It issues that applications operated in the cloud can slip out from the hand of its developer, as it can operate in environments where it is not immediately and intimately accessible, for example spanning world regions in data centres. Cloud-native applications expose three major types of metrics about themselves: data on their general performance (HTTP traffic, system resource metrics...), domain-specific data based on business knowledge (for example a medical diagnostic device's effectiveness at reading signals), and health and system logs (events such as startup, failure, scaling, shutdown...). The first two are the responsibility of the developer, the last one depends on the cloud operator. The amount of information that these metrics provide should be carefully planned. Depending on the utilisation of these applications, a huge amount of information could be dumped on the system, and managing distant applications depends heavily on the information it provides of itself.

Lastly, the security factor [60, Chapter 15] is aimed to fill the hole in the original methodology left by the topic. This is not as expansive as the other amendments, it mainly states

[16] See JetBrains Developer Survey 2020 [64].

[17] Posted in 2016, Horowitz's remarks have the chance of considering a more evolved cloud solutions ecosystem than Wiggins, originally in 2011 and his latest revision of the factors in 2017. This writer assumes that Wiggins had no intent on making big changes to his factors, seeing how deep an impact they had originally and the value of keeping them in their original state.

that in the face of general practice, security should not be a secondary consideration in the development process. It mentions authorisation and authentication as important factors to be implemented, if only on the most basic level. Role based access control (RBAC) with various standards and implementations[18] are cited as solutions for separating the service's functions based on the consumer's identity, as knowing who did what is important. The fact that these services will be operated across the world in data centres, in containers and accessed by the public or fellow developers through its endpoints makes inherent security vital.

Hoffman's remarks show the evolution around cloud-based application development and operation that had occurred since the original publication of the twelve factors. His additions are aimed at factors that might not have been as prominent at the time but emerged as ways for standardising cloud-based application development, a tendency shown by Horowitz's points as well, who tried applying the factors on the microservices-based architecture. It is important to note, however, that despite modifications and additions, the original twelve factors largely remain the same, and even if their conception was based on the personal experiences around Heroku's platform, Wiggins' concept for the ideal web application serves as a point of reference for modern, cloud-native application design.

3.4 Software Development Methodologies

Software development is the process of writing computer programs to fulfil the demands of a section of people, private consumers or professionals, by applying a set of technological tools and solutions to solve well-defined problems. This problem can range from small, concise services to large enterprise solutions serving thousands of employees with a set of defined, professional functionalities. Large development projects can produce thousands of lines of code and a single software product might be made up of different, standalone services. This necessitates teamwork, which in turn requires methodologies for those teams to follow in their work.

One of the main reasons for the existence of development methodologies is that while a simple script can be maintained by a single person, with growing numbers of lines of code and utilised technologies, the workload on a single developer increases as well. This means that the developer must have an increasing amount of technological knowledge and must keep up-to-date with the state of an ever-growing codebase. This can become untenable fairly easily. Development teams work on software products in order to pool skills and distribute workload. Using defined methodologies also ensures that common pitfalls will not sink the project as unforeseen problems emerge. The way in which tasks are defined, demands are measured, and deadlines are set intimately influence the way development work has to be managed. Something as basic as the scope of a task can influence how many resources have to be allocated to carrying out that task, how many hours should be spent on making sure, that the task is done in an operational and effective way.

Lots of other aspects come from these two basic points. Responsibility for the success of the project and for the distribution of resources, procedures for realigning project goals in case of changing circumstances, or methods for continually managing the process of different tasks are some of those. Development methodologies have been made so that development teams are prepared for these tasks and challenges, and are able to handle problems as they

[18] RBAC shortly means that users of a service are identified and collected in user groups who need to authenticate themselves to be authorised to use the resources and functions of said service. Their authorisation depends on their individual access control rules defined connected to their person or their group.

come along. However, these are influenced by the state of technology, and as the Internet has given immediate access to online services all over the world, development procedures have had to evolve as well.

3.4.1 Development Methodologies, Frameworks, Practices and Models

The summary of development methodologies could take up a work much larger than this thesis. This section only aims to introduce and elaborate on a set of defined modern development frameworks and practices, so the principles collected in this subsection are only in comparison with what will be written in Subsections 3.4.2 to 3.4.4 and as a short - and in no way definitive - historical summary.

Defining a way of organising development methodologies is hard in itself. Just the fact that there are methodologies, frameworks, and sets of principles makes it hard to compare them on the same basis. Winston Royce has written on the subject of developing large software designs and on his insights on how a process going through the steps of measuring requirements, analysis, design, coding, testing, and operation might prove fallible if not handled in a flexible and iterative manner [71]. That might have falsely provided the basis for the well-known waterfall-model [66, 50, 74]. This model, in its simplest terms, describes the development process as different phases whose products flow into the next phase. After this process is done, based on remarks and experiences, the whole process is repeated until the final product is done. This model can conjure the picture of a hobby application, rather than a long-term development project. That might be because, according to [66, 50, 74], the model might only have been a straw-man term and intended to be used exactly for the opposite as it has been over the years: a counter to the thought that a rigid, procedural cycle can effectively create good software.

The waterfall model is an example of how hard it can be to pinpoint exact models and frameworks, as they can evolve and change across development teams and cultures. It also exemplifies standard procedures and how inflexible methods cause problems, or at least perceived problems[19], in the development of software. As the final parts of this section, Subsections 3.4.2 to 3.4.4 will detail aspects better connected to this thesis, one paradigm and one methodology will be shortly introduced here as a primer, as, in this writer's opinion, they have a strong connection to those aspects and influence the developer community greatly.

Agile Development and Extreme Programming

The philosophy around Agile has been first codified in the Agile Manifesto [46]. The four core values are: "Individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; responding to change over following a plan". The manifesto also states an order of importance, "while there is value in the items on the right, we value the items on the left more". The authors of the manifesto intended to provide a new philosophy for approaching software development. Looking at the four core values, the agile philosophy states, in summary, that development work should be defined by the people carrying out the work and using the product instead of the processes and methodologies that these people use while doing their work. This is intentionally vague, as the signatories were of several, differing opinions. The main intent of the manifesto was to provide a frame of thought

[19] Cited sources allude to some of these problems being exaggerated on purpose to be put in opposition of other development approaches.

that would free development work from the strict processes that it has been confined in by traditional structures [1].

The Agile Alliance is an organisation that has grown from the manifesto and advocates practices and values associated with agile methodologies. They have elaborated on the original manifesto and defined twelve points that describe the agile philosophy, verbatim [3]:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximising the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organising teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

It is clear that these points are based innately on the four core principles; however, they allow better interpretation for the actual practices and methodologies that want to conform to these tenets. One of the main messages here is that customers and developers are people, and their needs have to be considered first and foremost. The requirements, processes and measures that lead their work should be defined by themselves, according to how they see fit to work on their individual products. This needs flexibility and the desire to change. This desire should not be self-serving though. Changes should be driven by the changing environment the product is developed for. To actually see this change, developers and customers need to interact closely, promoting cooperation. Finally, teams should be self-organising and should have the possibility of retrospection, to change the way they conduct their work based on their experiences. The constantly stated continuity requirement - as in continuous delivery of software and the frequency of that delivery - and openness to changing specifications also echo the basis for the practices further detailed in Subsections 3.4.2 to 3.4.4.

Martin Fowler has written in depth about what he thinks the agile methodologies bring to software development [53]. He concentrates on these methodologies' adaptability and

people-orientation, in parallel with the manifesto's message. These insights are valuable elaborations on the previously mentioned aspects; however, his listing of agile methodologies and thoughts on who should adopt these are just as much informative. Among the listed methodologies he mentions Extreme Programming (XP), the tenets of which were defined by Kent Beck in Extreme Programming Explained [44]. As of Fowler, XP aims to bridge certain practices and abstract values with principles that help synergise the whole methodology into a frame that can be used to drive software development. By Michele Marchesi [65], XP has seen a big revision over the years, the number of its values, practices and principles has increased[20]. Methodologies associated with the agile philosophy evolve dynamically, and rightly so, as the philosophy itself encourages change. The increase of practices shows the inclusion of experiences of those, who utilise XP. Three of those practices with close connection to topics in latter parts of this section are the primary practice of continuous integration [65, Page 5-6], and the corollary practices of a single code base and daily deployment [65, Page 6]. The evolution of XP is a good example of the changes the agile movement advocates. However, as of Fowler [53], adopting agile practices is not a solution to everything. The difficulty in their adoption is actually the human nature of the changes that have to be made in a development team's day-to-day work, and as such, it should not be attempted in communities where the members themselves do not want to adopt them.

XP is just an example of the evolution of development methodologies, and just an example from all the aspects of philosophies that the development community uses in their everyday work. However, methodologies such as XP, and in essence the agile philosophy, advocated for changes that have defined a new way of delivering software solutions to customers. Some of those have influenced this thesis and they will be detailed in the following subsections.

3.4.2 Continuous Practices

Continuous practices is used in [72] to refer to a set of practices that can be mentioned as part of "continuous software engineering". Those three practices are Continuous Integration, Continuous Delivery and Continuous Deployment. According to [72, Chapter II], these practices are aimed at the accelerated development and stable, frequent delivery of software products without compromising quality using automation in all the places where manual tasks can be substituted with tools, while bringing development and operations teams closer in cooperation for the seamless functioning of their provided software services. The mentioned practices enable these teams to work together as they make links between the workflows of each other. Additionally, frequent exchanges of feedback between developer and customer must ensure that experiences and errors are integrated into the development work so that the software will always be reactive to its users' demands.

Continuous Integration (CI) - while being a primary practice for XP - is a main topic for Humble and Farley [63, Chapter 3]. Perhaps the most important idea they describe is that while using continuous integration, the developed software is always proven to work, it is at a working stage all the time. Without it, the proof of it working comes from occasionally performed testing or integration. However, by committing every change the development team makes to the codebase - which inherently implies the use of version control systems - the automated build, unit and integration tests will be performed on

[20] As of Marchesi [65, Page 1], Beck defined four values, fifteen basic principles, and twelve practices in the first edition of his book [44, Through Marchesi, 2005]. In the second edition [45, Through Marchesi, 2005], there are five values, fourteen principles, thirteen primary practices and eleven corollary practices (24 practices in all).

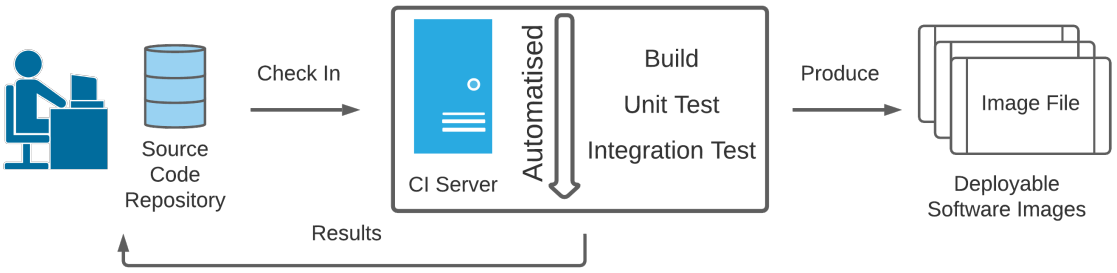

Figure 3.7: Illustration of an example workflow implementing Continuous Delivery.

each build and any errors that the changes might introduce to the codebase is apparent. At this stage, Humble and Farley state that the fixing of this problem actually preempts any other task the team might have. By always being up to date on the working state of the software, it is also cheaper for the team to fix these bugs. For them, CI is possibly on the same level of importance as using version control in development projects. Common aspects of CI include [63, Chapter 3] [72, Chapter II] [55]: use of a single, versioned source code repository; multiple check ins from development work to the source code mainline[21] daily; assurance, that at any time, the mainline software is in a working state; automated build and testing; fast build and test times so that frequent check ins do not block the workflow; common access to the whole codebase for every team member; and frequent communication on and well-defined responsibilities for the state of the software between team members. This all enables rapid development of software while ensuring the best quality and seamless cooperation at all times.

The following two practices cannot be separated as well as they can be from CI, so they will be detailed together. They are Continuous Delivery (CDel) and Continuous Deployment (CDep)[22]. While CI aims to streamline the development process and ensure that the software is always at a working state, CDel ensures that this software is always at a production-ready state by using CI and automation to carry out builds and testing, as shown in Figure 3.7, reducing risks connected to software deployment, lowering cost and accelerating feedback between developer and customer. CDep connects with this process and, using automation, steadily deploys a production-ready software to a production environment, like the one in Figure 3.8. With CDel, deployment is a manual decision, CDep carries that out automatically [72, Chapter II].

CDel is inherently an important business decision as well as a technical one. When software is deployed, it starts being an active business asset. This means that its deployment cycle and that cycle's success also affects business considerations. CDel ensures that each step in the deployment is traceable, different business actors can build their own version of the deployable software and see how it behaves, and different CDel tools allow role-based access to their processes. Automation also minimises human error in the deployment and enables thorough testing. For this reason, Humble and Farley [63, Chapter 15] wrote

[21] Multiple phrases could be used to describe the version controlled workflow. To avoid confusion, the basic workflow is that there is a main point which is always the most current state of the software, and all changes made to it are written on another version, which is "branched" from the main version. Then, after all the required changes have been made and possibly checks are applied, such as testing, it is merged into the main version. This causes another, more current version to be at the main point and further development is conducted from that point onward. A specific set of principles is called gitflow [51].

[22] In other descriptions, these two are all abbreviated as CD; however, for the sake of clarity, more descriptive abbreviations are used here.

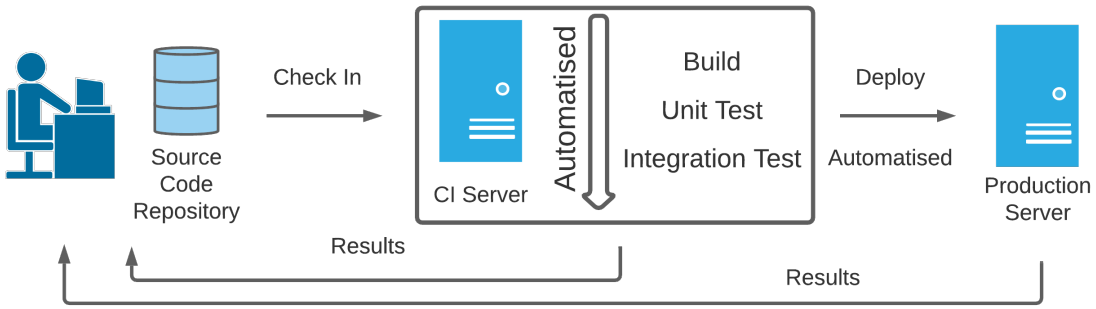

Figure 3.8: Illustration of an example workflow implementing Continuous Deployment.

in depth about CDel management, and the fact that they heavily pronounce business considerations next to technical ones shows how much these two fields connect considering software delivery. In summary, using CDel practices ensures that "the current development version of the software can be deployed into production at a moment's notice - and nobody would bat an eyelid, let alone panic" [56].

CDep builds on CDel practices and they must be present for CDep to be possible [56]. Research also suggests, that CDep might not be suitable for all business considerations [72, Chapter II]. Nevertheless, if practices are in place, CDep allows incremental changes to the software to be continually applied to software in production by automatically going through deployment pipelines where different processes are run to ensure that the software actually conforms to requirements, such as builds, tests, or package publishing. A basic overview is illustrated in Figure 3.9 [72, Figure 1] [73, 68].

To support adoption of CI/CDel/CDep, a huge number of tools have been designed to help in different stages of the deployment pipeline[23]. A large permutation of these can be used to make an integration and delivery system that enables the acceleration of software development and the increasing cooperation between teams responsible for said software. The evolution of these practices was precipitated by the agile philosophy, as their tenets for the continuous development of quality assured software needed defined methods and tools that could be integrated into actual development work. Today, these practices have an important place in the developer community. In a survey, 44% of developers who use Docker images said they used a CI/CDel/CDep framework [64] to deploy those images, 61% of those respondents who use cloud hosting services said they use CI/CDel/CDep frameworks for deployment, and 45% of all respondents said they used CI or CDel/CDep tools regularly[24]. With cloud computing's pace of evolution, these practices and tools are even more important, as opportunities for cloud-native operation rise. If software development is faster and easier when companies do not have to own all the resources needed, and computing resources are available on-demand, practices that enable this way of development will be popular, and this shows great promise for continuous practices.

[23]For a list of these, see [72, Figure 4] and a collection of tools at [9] as part of the larger DevOps ecosystem.

[24]Important side note: the survey uses abbreviations in places, while full phrases in others. As such, this thesis makes the assumption that CDel and CDep tools constitute "CD" as it is used by the surveyors.

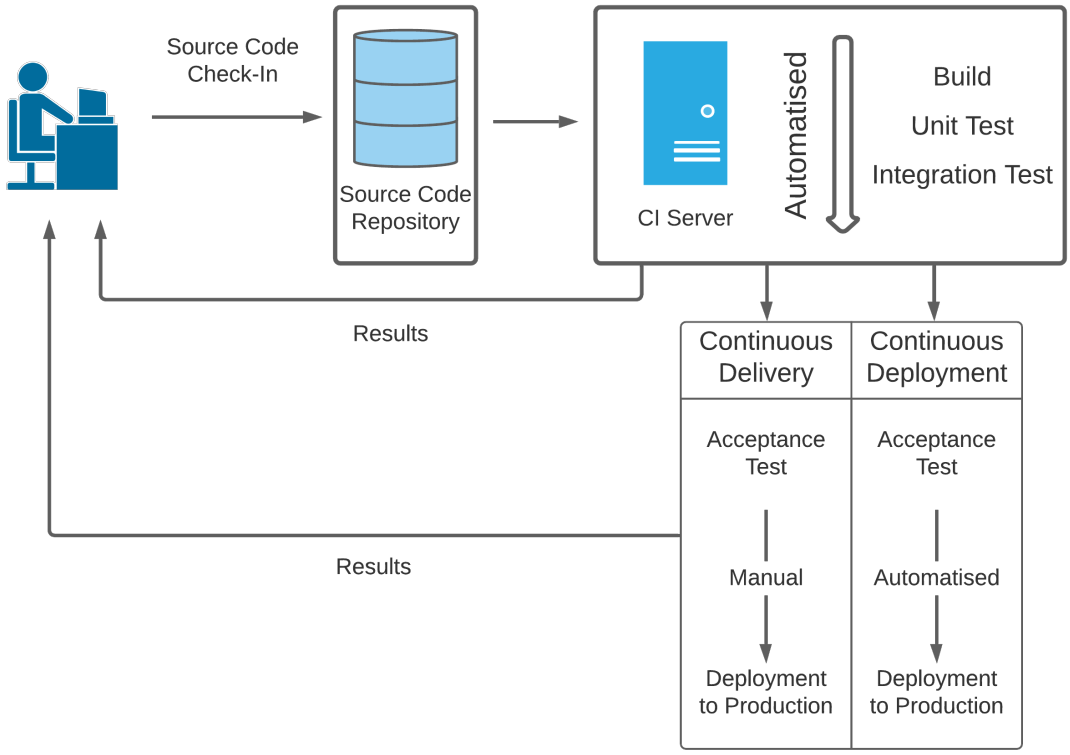

Figure 3.9: A basic CI/CDel/CDep workflow illustrated.

3.4.3 DevOps

While detailing the short history of development methodologies, it has been stated, that it is difficult to pinpoint them, as a lot of those change across different cultures and teams. The evolution of these methodologies shows a wide curve. From methods of the middle of the 20th century, such as Royce's iterative model, to the extremely adaptive and granular practices of today, which were greatly influenced by the signatories of the Agile Manifesto, there has been a steady growth of software development culture. As development work reaches the level of maturity where philosophies emerge about how it should be done best, some principles begin living their own life, abstracted from the actual practices that developers use day-to-day. DevOps is such a framework.

The Gartner Glossary defines DevOps as "a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach", which seeks to improve cooperation among teams responsible for the service's lifecycle, utilising automation technology [12]. Some of the biggest IT industry leaders echo these as well [2, 20, 30]. Culture is an important term, as it implies that DevOps aims to be more than a collection of tools and practices. An interesting compilation has been published by the DevOps Research and Assessment team of the capabilities that they consider in relation to a well-working DevOps culture, defined in Table 3.1 [49]. This thesis considers this the best defined set of principles and, as such, considers it to be what DevOps stands for[25].

[25]With the caveat: DevOps is not definitive. There are many descriptions and frameworks advertised as DevOps. It cannot go unstated that this decision is necessarily subjective, as is the essence of DevOps itself.

Technical	Process	Measurement	Cultural
Version control	Team experimentation	Monitoring and observability	Job satisfaction
Continuous integration	Streamlining change approval	Monitoring systems to inform business decisions	Westrum organisational culture
Deployment automation	Customer feedback	Proactive failure notification	Transformational leadership
Trunk-based development	Visibility of work in the value stream	Work in process limits	Learning culture
Continuous testing	Working in small batches	Visual management capabilities	
Continuous delivery			
Architecture			
Cloud infrastructure			
Test data management			
Empowering teams to choose tools			
Shifting left on security			
Database change management			
Code maintainability			

Table 3.1: DevOps capabilities defined by DORA

Without the need for definitive description of each capability[26], it is clear that most of these - by their name - are closely related to aspects previously defined in this chapter from Section 3.1 to Subsection 3.4.2. Most of the technical capabilities enunciate practices that conform to the continuous practices (continuous processes, automation, testing and code maintainability) detailed in Subsection 3.4.2 and also have aspects in common with the goals of the microservices architecture (free experimentation with tools on team level), elaborated on in Section 3.1. The process capabilities share the same message as the values stated in relation to the agile philosophy detailed in Subsection 3.4.1. The practices listed under measurements bring the continuous observation of the software into consideration, also an important part of continuous practices; however, it also includes aspects about the amount of work done at once by the team in question and the ability of self-measurement (think visual management tools, like storyboards where specifications are detailed in a visual and everyday manner). Steady organisation of workload is also a part of DevOps culture. Finally, the capabilities under culture bring the DevOps circle to a close and give the framework its own essence.

While, as detailed, most of the previous capabilities have a lot in common with other, parallel methodologies and practices, DevOps brings them all together under a far-reaching, cultural shift. Leadership and cooperation constitute the basic tenets of DevOps, its main objective is to bring together teams that previously worked in silos, meaning disconnected from each other in terms of technology, management and communication. The amount of disparate processes it means to connect is shown in Figure 3.10 [8], a commonly used illustration of the DevOps workflow. The name DevOps is a sign of that as well, Dev stands for development and Ops stands for operations. This shift must start with the people in these teams, nevertheless; they have to be the ones driving these changes and

[26]Detailed description of each can be found following the citation.

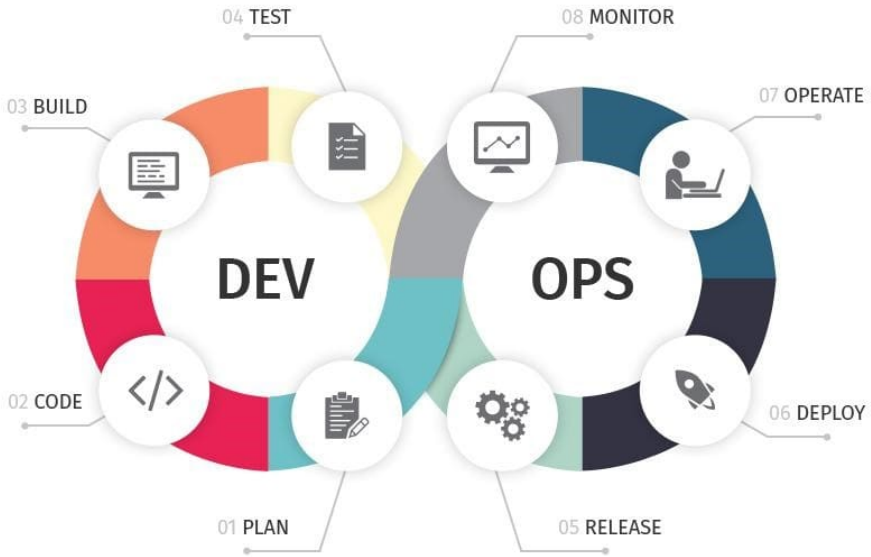

Figure 3.10: The DevOps infinite workflow cycle.

applying the methods and practices that DevOps entails. As such, the success of DevOps lies more with those implementing it rather than the technical environment.

The evolution of DevOps and its adoption has brought about a scene of other "Ops"-es, their goal being to bring the cultural aspects of DevOps to other technical areas. Security has become increasingly important as software is delivered more and more rapidly, and this brought on DevSecOps, which aims to intersect security with the processes of DevOps [31]. Another example of this is FinOps, which brings financial accountability to the fiscal management of cloud services [10]. A common theme is to bring these specialised teams out of their silos and promote close connection and cooperation with other teams. It can be deduced that DevOps has not only brought on a cultural shift in development and operations teams but has given rise to a framework which is technology and speciality agnostic. Its tenets are being applied to different areas that have connection to IT operations. It is hard to properly adhere to the DevOps mentality; however, it characterises modern software development culture and plays a big part in the philosophy of the IT community.

3.4.4 GitOps

In the end of Subsection 3.4.3, it has been stated, that DevOps has given rise to a number of other frameworks that seek to utilise the cultural and technological practices of DevOps in specific areas. GitOps is one such framework. It will be detailed further in this subsection, as it connects in many places with the topics of this thesis, such as the operation of a cloud-based application, the supporting cloud infrastructure, and cloud-native best practices.

GitOps was described by Alexis Richardson in 2017 at Weaveworks [33, 16] and it has seen steady growth since its inception. In summary, GitOps is "[an] operating model for Kubernetes and other cloud-native technologies, providing a set of best practices that unify

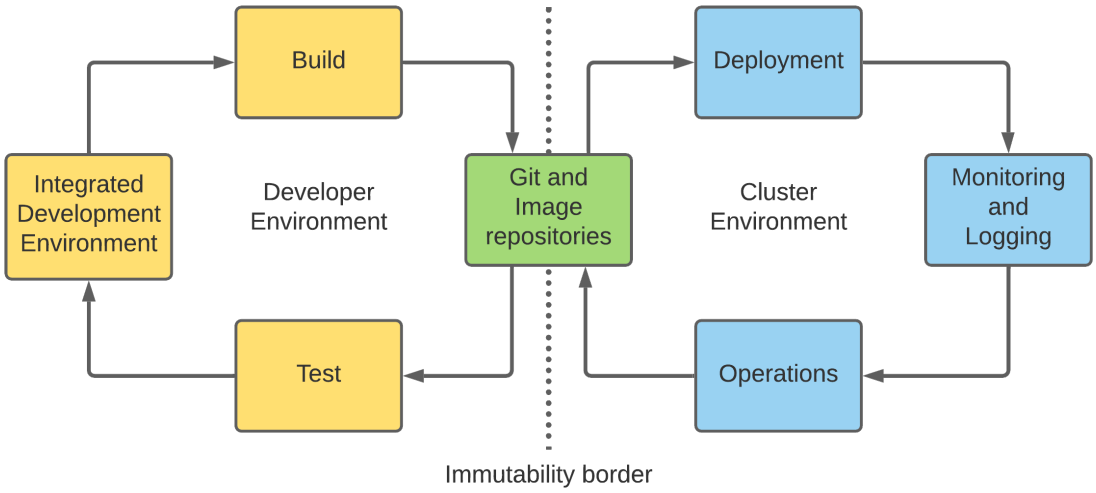

Figure 3.11: Illustration of how Git connects the development and operations workflows together.

deployment, management and monitoring for containerised clusters and applications. A path towards a developer experience for managing applications; where end-to-end CI/CD pipelines and Git workflows are applied to both operations, and development" [33]. This flow is illustrated in Figure 3.11 [13], as well as the immutability border, which denotes the two domains whose changes must not directly influence the work performed by the other, only through changes in Git.

What is immediately evident is that GitOps was made with the version control system Git and the container orchestration tool Kubernetes in mind; however, by their own admission, its principles are able to be used with any other tools that serve a similar purpose. Kubernetes is used as it allows for the declarative description of the desire state of cloud resources, as detailed in Chapter 2, which is one of the basic principles. GitOps aims to give developers tools that enable them to define their production environments and operate them inside the same workflow that they use for development work. This does not mean that operators are excluded from this work, but rather means that they are part of the same workflow and can use the same set of tools for its automation. "The core idea of GitOps is having a Git repository that always contains declarative descriptions of the infrastructure currently desired in the production environment and an automated process to make the production environment match the described state in the repository. If you want to deploy a new application or update an existing one, you only need to update the repository - the automated process handles everything else" [16]. By keeping the declaration of the desired environment state inside a version controlled repository, multiple benefits are gained:

- there is a clear audit log of changes applied to the environment and their reasoning through easily understandable commit messages;
- as changes are introduced as commits that iterate over a moment in the repository's state, those changes can be easily and quickly reverted as well in case of failure or other problems, since the system keeps the description of the previous states in its history;

- all this is centralised, there can always be a single source of truth about the complete description of the desired state, allowing information to be freely accessible by teams working on their products.

In order to integrate GitOps practices in a development workflow, four principles must be followed [33]:

- The entire environment must be described declaratively.
  This ensures that the actual state is always the same as the desired state, as it does not specify open-ended instructions but rather facts that must be true about the environment. Of course, it cannot be said that the actual state is always the same; however, the declarative description allows for differences to be noticed and acted upon.

- The canonical state of the environment must be stored in a version controlled repository.
  This is a separate repository to all other in the development workflow. Using the capabilities of systems such as Git, the main source of truth is a repository that can only be modified through a well-defined and secure process, and its history enables reasoning of changes in the environment and reverting those changes if need be.

- The changes made in the environment repository must be automatically applied to the environment.
  By automating these processes, developers can be sure, that committing changes to the repository will mean, that the underlying system represents the state declared in the repository. This serves to connect the development and operation of the software.

- Monitoring solutions must be used to ensure correctness and alert on divergence.
  Information must be available on the correct operation of the environment. This can range from the health of different components to the correct implementation of specification. If Kubernetes is taken as example, individual Pods might be self-healing; however, these aspects should still be monitored for complete coverage. What must be monitored though is the correct configuration of the environment to rule out human error and misconfiguration.

Using GitOps principles in the development workflow allows for developers to have a larger role in the operation of their software. By using version control systems, Git as commonly referenced by [33, 16], familiar processes can be applied to infrastructure configuration.

This operation enables two methods of applying changes in the repository to the environment, the push-based and the pull-based approach. This describes the way in which changes are deployed to the environment. In the push-based approach, changes committed to the repository are then "pushed" to the environment in a pipeline that automates the process of applying the changes. This means two things: there has to be a separate process for the actual application of the desired changes, and there is no inherent method for the environment to notice or propagate changes back to the repository if the environment's state diverges from that described in the repository. There is a bit of a disconnect between the repository and the environment, which is bridged by the pipeline that handles the state of the environment. Figure 3.12 shows an illustration of this approach [15].

The pull-based method inverts how the changes in the declarations are deployed to the environment. In this model, an operator software acts as the overseer of the environment

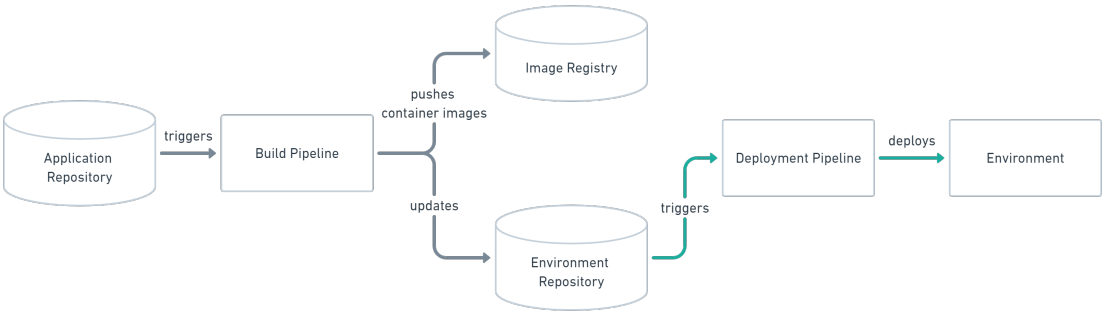

Figure 3.12: Example of the push-based deployment approach

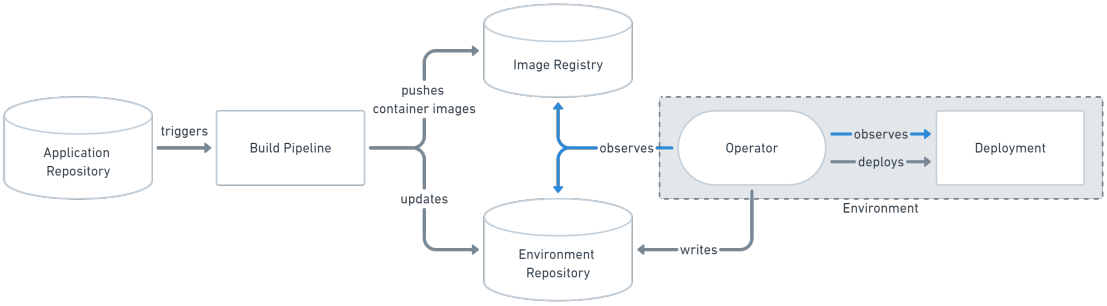

Figure 3.13: Example of the pull-based deployment approach

and it makes sure that the environment resembles the desired state stored in the repository. This operator, firstly, monitors the repository and deploys changes. Secondly, it ensures that deviation caused by outside sources are rolled back, so that the environment does not divert from the state in the repository. This method necessitates the presence of such an operator software; however, this is the most secure way of automating the process, as no undue authorisation has to be given to third-parties, the operator is part of the infrastructure and changes are carried out by the environment itself. Kubernetes' example would be the application of changed manifest files through kubectl, after which Kubernetes ensures the desired operation of its cluster. Figure 3.13 illustrates this approach [14]

Whichever method is used, the automated deployment of environments enables development teams to rapidly develop and deploy software in multiple stages of the software's lifecycle. The fact that everything is "logged" in a sense in the version control system enables rollbacks between versions of the environment and the observation of reasons behind changes at a given point. Development team members do not have to directly interact with the environment, so no access has to be given, security can be better implemented, especially if the environment operator can act on its own as part of the infrastructure. Finally, these practices allow better communication between teams involved with developing software solutions, as all steps are reproducible through the version control system, everybody has access to a single source of the current stage of development and the disparate processes of development and operations teams are easily connected with each other through common tools.

3.5 Cloud-Native Software Development

In Chapter 3, different aspects of software development have been elaborated on. With the findings in these sections, Section 3.5 aims to define cloud-native development, as it relates to the development work conducted in this thesis. By defining what part these findings play in cloud-native development, a framework will be in place for later parts to reference, when the actual design and development work is detailed.

Cloud-native technologies, as defined by the Cloud Native Computing Foundation (CNCF) "empower organisations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil" [6]. Decomposing this definition allows for the inclusion of previous topics.

Cloud-native technologies run scalable application in dynamic cloud environments with declarative APIs. Chapter 2 details Kubernetes, an orchestration tool for the dynamic provision of container-based applications. If an application aims to be cloud-native, its developers must make sure that it can easily be operated by tools that abstract away the processes of cloud resource allocation and conform to the standardised processes that these tools support. An example would be how an application is containerised and made available for Kubernetes to access. In this thesis, this method is the usage of public container repositories and the definition of their access in the Kubernetes configuration.

Applications are developed in a cloud-native manner by using microservices and containerisation technology to make loosely coupled systems. For the scaling of the application to be the most efficient, small, independent microservices are best used, as these can be scaled horizontally the easiest. Horizontal scaling means that increasing load is handled by initiating more instances of a certain service. This is more cost-efficient compared to vertical scaling, which means that more resources are allocated to an already running instance. Section 3.1 details the architectural aspects of microservices development, while Section 3.3 elaborates on the manner in which web services should be developed. While the former is an architectural pattern which defines a higher level in the design process, the latter contains guidelines on how an application should be designed on a deeper, development-specific level. Section 3.2 details an architectural pattern that has evolved in connection with the challenges of keeping data available and consistent in distributed systems necessitated by such an approach.

Cloud-native technologies enable systems to be resilient, manageable, and observable. Resilience is the attribute that defines how well a cloud-based system performs under dynamically changing load. Specific processes in Kubernetes and the disposable nature of a web application, as defined by Section 3.3, enable the easy build up and tear down of system resources depending on usage. The manageability of these systems is ensured by the abstractions provided by Kubernetes and by following practices that streamline infrastructure management, such as those defined in Subsections 3.4.2 to 3.4.4. The observability of such systems is achieved by careful consideration of logs and metrics in the design and development process, as defined in Section 3.3, and the practical application of observability solutions, such as those defined in later parts about this thesis' development work.

These aspects, combined with robust automation, allow high-impact, quick, and predictable changes to be made to systems running in a cloud-native environment with minimal toil. As software development accelerates, human error has to be ruled out by putting repetitive processes through testable, automated pipelines. The deployment of cloud-native infrastructures is such a process, and its automation allows for engineers to constantly change details in the environment without much impact on how those changes are deployed in a working cloud infrastructure. This needs a consistent effort on a technological and human level as well. The way in which this changed the tools and culture around software development is detailed in the subsections of Section 3.4. Additional attention is given to automation in Subsections 3.4.2 to 3.4.4. Subsection 3.4.4 elaborates in more detail on how infrastructure management changed due to these developments.

Cloud-native development seeks to utilise the growth in cloud computing technology. The details in this summary section give a shallow overview on how cloud-native development can be perceived from the viewpoint of this thesis. However, its growing landscape is huge, and much more could be written about it. Nevertheless, Chapters 2 to 3 of this thesis give an overview which will be used to describe the practices that have been used in this thesis for the development of a cloud-native application.

Chapter 4

Practical Aspects of Cloud-Native Software Development

In Chapter 3, the fundamentals of cloud-native software development have been detailed, while in Chapter 2, Kubernetes has been introduced as a solution for orchestrating container-based applications in a cloud environment. In Chapter 4, this thesis' practical work will be introduced, which aims to demonstrate the use of the aforementioned practices and methodologies in an actual, working system.

Firstly, the microservices-based software will be documented, which is to be used in performing testing of the cloud environment it will be deployed in. After that, the automated continuous development environment will be drawn up concerning the CI/CDep solutions and versioning system that will be used. Thirdly, the cloud environment's basic structure will be detailed, as regards the workflow that this thesis employs. This means that some aspects of this environment will be treated as outsourced, not pertaining to the scope of this writing. Considering this, the documentation of the Kubernetes cluster will be provided, and its services that are used by the deployed application. Following that, a short documentation will be written about the monitoring, logging and alerting solutions that are used to operate the deployed software. Finally, a use case test and its findings will be presented which serve to show the operation of the employed technologies in practice. In all these sections, where applicable, parallels will be drawn between the implementation of the technological solutions and the practices introduced in Chapters 2 to 3.

4.1 Cloud-Native Software Solution

The software that has been developed as part of this thesis' work is a simple inventory and shopping basket application. As it is primarily used for testing purposes, its design is more focused on functionality than being well-rounded. As such, it only performs to the specific needs that the used methodologies and the surrounding cloud environment has. Its architectural composition was planned so that the cloud-native development workflow, and cloud-native operation are easily implemented.

4.1.1 Software Design

The basic setup contains two microservices. One is an inventory service, whose task is to persist items that the store needs to keep track off. This includes the creation of

```
"""The @ decorator provides the HTTP and path settings
   for the function.
   """
@app.route("/", methods=["GET"])
def inventory_process(error=None):
    """The HTTP requests counter's specific label
     gets incremented by each call.
     """
    http_counter_metric.labels(method="GET", endpoint="/").inc()
    """The server aggregates the items in memory and prints
     responds with the main page HTML, with variables set.
     Error messages are provided when needed.
     """
    inventory = {
        item.name: item.amount for item in \
        inventory_web_interface.inventory.values()
    }
    return render_template("inventory_page.html",
                           inventory=inventory,
                           error=error)
```

Listing 3: The function defining the actions that must be taken by the application when accessing the "/" path exposed by the Flask server.

these items, their stock-keeping, and their deletion from the store. The other service is an order service, which provides the functionalities of a shopping basket. This service is bare bones, other than creating basket instances, represented as event streams in the data store, and adding arbitrarily provided items, it does not provide other functions. These microservices use an event store, the EventStoreDB, as a persistent database. This is where the functions of each service write the actions they perform by sending the logical results of these operations as events.

Each microservice can be described by using aspects of the three layered, domain-, and event-driven architectures. A service contains a primitive user interface, a layer that implements business logic, and a layer that translates logical operations - or commands, as it is used in this design - into events, which are transmitted to the event store. The UI serves as a point of interaction between the software and the user. The business logic layer performs the following functions: contains the logical representation of the objects whose state are described by the sequence of events that are emitted from the service; executes logical checks on user commands, so that they conform to the logic that governs the domain the service implements; and translates commands into domain terms that can be used to construct events. The event layer uses information provided by the business logic layer to generate events, and makes sure that these events are written to the event store, to the appropriate stream, in the appropriate manner.

Both services have a web server framework, Flask [11], integrated. This enables the exposition of the service's functions through paths defined in a separate Python module by reserving a port for network communication. It allows performing actions defined in Python functions that correlate to paths that can be reached on the internet, with different settings available, such as the allowed HTTP methods. The "/" path of the

```
"""This function handles the execution of an Event object."""
def execute(self):
        """The event counter is incremented on each execution."""
        Event.event_counter_metric.inc()
        """The event receives a unique ID, and is written
       to the event store stream, identified by the
       aggregate ID set during the creation of the event.
       """
        es_id = uuid.uuid4()
        headers = {
            "Content-Type": "application/json",
            "ES-EventType": self.event_type.value,
            "ES-EventID": str(es_id),
        }
        requests.post(
            f"{os.getenv('EVENTSTORE_STREAM_URL')}/{self.aggregate_id}",
            data=json.dumps(self.data),
            headers=headers,
        )
```

Listing 4: Excerpt from the codebase that handles the sending of events through the HTTP API by using the requests package.

inventory service can be seen in Listing 3. For communicating with the EventStoreDB through its HTTP API, the requests package [32] is used, as shown in Listing 4.

As mentioned, the service keeps the state stored in the event store in logical Aggregate objects, which summarise the sequence of events into the momentary, actual state. This object is the point of contact between the event-based system and the user intent[1], as this is where a command submitted to the logical object is translated into an event emitted to the specified event stream, which is defined by the specific identity of the object. For example, in the inventory service, a command expressible as "increase X stock by 20" is submitted to the X Aggregate object, and it will egress an event that states "Stock increased by 20" to the X stream of the event store.

This Aggregate has been designed with additional event sourcing capabilities in mind, those that are not necessary for testing purposes, such as the capability to:

- keep the version of the Aggregate object; this enables the application to ensure data consistency by comparing the versions of the logical and the stored state
- reverse translate the events that it might receive; which enables the logical effects of a parsed state to be applied to the logical Aggregate
- iterate over an event stream's events and apply their effects in sequence; providing the ability to recreate a sequence of events in a fresh Aggregate object.

The implementation of these shared and specific capabilities are shown in Listings 5 and 6.

[1] As noted in Section 3.2.1, intent in this context is simply a turn of phrase.

```python
class Aggregate:
    def __init__(self):
        """The ID of the Aggregate correlates
        with the event stream it represents.
        """
        self.aggregate_id = uuid.uuid4()
        """The version number - used to identify individual events in a stream -
        enables consistency checks.
        """
        self.version = 0

    def apply_event_effects_to_aggregate(self, event_json):
        """Event sourcing function not implemented."""
        pass

    def raise_event(self, event):
        """Execution function for easier reading while handling Aggregates."""
        event.execute()

    def load_up(self):
        """Function enabling the processing of an entire event stream.
        The ID of the stream is defined by self.aggregate_id.
        """
        version = self.version
        """Events can be read in order in the stream from the version point."""
        while True:
            request = requests.get(
                (f"{os.getenv('EVENTSTORE_STREAM_URL')}" +
                 f"/{self.aggregate_id}/{version}"),
                headers={"Accept": "application/vnd.eventstore.atom+json"},
            )
            """In sequential reading of a stream, when there are no more events,
            a non-200 message is received, signaling the end of the stream.
            """
            if request.status_code == 200:
                self.apply_event_effects_to_aggregate(request.json())
                version += 1
            else:
                break
        self.version = version
```

Listing 5: The base Aggregate class.

```python
class ProductStockAggregate(Aggregate):
    def __init__(self, name, amount):
        super().__init__()
        self.name = name
        self.amount = amount
        """Status is a one-step indicator of an event stream's current status."""
        self.status = PRODUCT_STOCK_STATUS.INACTIVE

    """Function enables parsing of domain-specific events in event streams."""
    def apply_event_effects_to_aggregate(self, event_json):
        event_type = event_json["summary"]
        data_dict = event_json["content"]["data"]

        if event_type is INVENTORY_EVENT_TYPE.StockCreated.value:
            self.status = PRODUCT_STOCK_STATUS.CREATED
        elif event_type is INVENTORY_EVENT_TYPE.StockAdded.value:
            self.amount += data_dict["amount"]
        elif event_type is INVENTORY_EVENT_TYPE.StockSubtracted.value:
            self.amount -= data_dict["amount"]
        elif event_type is INVENTORY_EVENT_TYPE.StockDeleted.value:
            self.status = PRODUCT_STOCK_STATUS.DELETED

    """Function handles specific event sending and keeps status up-to-date."""
    def create_stock(self, reason):
        payload = {"reason": reason}
        self.raise_event(
            domain_events.StockEvent(
                INVENTORY_EVENT_TYPE.StockCreated, self.aggregate_id, payload
            )
        )
        self.version += 1
        self.status = PRODUCT_STOCK_STATUS.CREATED
```

Listing 6: Excerpt from product_stock_aggregate in the inventory microservice.

Event Stream 'X'

self | first | previous | metadata

Event #	Name	Type
2	2@X	StockAddedEvent

Data
{
"amount": "20"
}		

1	1@X	StockCreatedEvent

Data
{
"reason": "administratively created"
}		

0	0@X	ProductCreatedEvent

Data
{
"name": "X",
"price": "200",
"currency": "HUF"
}		

Figure 4.1: A sample event stream in EventStoreDB of item X with 20 stock.

To support this methodology, the events emitted by the services have been designed in such a way, that they allow for sequential reading and application to a logical summary. Events have a type, it describes what the event means for the stream it is appended to. There is a genesis event for all event streams[2]. After the stream is created, any event appended to it details a single occurrence, the effect that it has on the logical whole. If product X has been created and 20 stock has been added to it, the sequence of events in stream X can be seen in Figure 4.1. As visible, it contains statements as event types in the past tense, detailing a single occurrence. The payload of the event contains details that describe how that specific event has passed. The genesis event is the "ProductCreatedEvent" that includes the product details, the secondary genesis event is the "StockCreatedEvent" with the reason for the creation of the stock. After appending 20 stock, the event "StockAddedEvent" describes this with the details of the occurrence, the amount of the addition. Individually applied, these provide no information on state; however, stringed together in a sequence, they gain meaning.

In order for logging and monitoring to be implemented, Python's logging module and the Prometheus instrumentation client library [22] was used in the microservices. The services emit logs to the standard output, which is the method the log collection solution, Promtail [28], scrapes these entries with. These are then aggregated by Loki [18]. The metrics that measure quantifiable data are exposed at a specific path - "/metrics" - in order for Prometheus [25], the metric collection solution, to collect them. These solutions are further explained in their own sections, the instrumentation of these functions will be detailed here.

4.1.2 Logging

Logging is implemented at certain parts in the code, where meaningful messages help understand the functioning of the application. Python's logging module allows for individual logger instances in each of its modules; however, in this application, the root logger instance is sufficient. It is initiated in the package initialisation file of each service that runs the first time a Python package is executed. Where applicable, log messages of different priority are created and sent to the standard output. Two priority levels are used: INFO and ERROR. The sole informational log message is when the application is being started and when it has finished starting up - because of Python's unique import structure, this happens when the ___init___.py file is run at package execution and when all the files are completely initialised through imports at the end of views.py[3], exemplified in Figure 4.2. Error messages are logged when exceptions are encountered, and there are two main categories in use: exceptions relating to the use of the requests package for network connections to the event store, which is shown in Listing 7, and when exceptions are encountered through the functioning of the application itself. These message are for-

[2]In fact, for the inventory, there are two genesis events, as the product and the stock are created separately but handled in the same stream.

[3]In Python, a single .py file, called a module, can import other modules to access its features. When this happens, all the initialisation tasks described in the imported module are run before moving on to further parts of the original code. This means that, if modules are imported hierarchically down the line of modules, all modules that take part in the functioning of an application will be initialised when the script that first imports the starting modules are finished. Thus, if the package execution starts with ___init___.py and the log message "app is being initialised" is emitted, it imports views.py, which in turn imports all the modules that are part of the application and all the imports eventually return to views.py, and the code reaches the end of views.py where it logs the "app has been configured successfully" message, it means that there were no errors encountered while interpreting all the modules that the application needs to run.

```
from . import prom_logs

"""Enumeration of variables related to errors expected to occur"""
expected_main_error_types = [
    requests.exceptions.ConnectTimeout,
    """And on..."""
]
error_logging_messages = {
    "ConnectTimeout": f"network timeout error" +
        f"while connecting to {os.getenv('EVENTSTORE_STREAM_URL')}",
    """And on..."""
}
error_logging_error_codes = {
    "ConnectTimeout": 502,
    """And on..."""
}
error_logging_metrics = {
    "ConnectTimeout":
        prom_logs.performance_metrics["network_timeout_error_counter"],
    """And on..."""
}
def log_and_return_connection_error_response(e):
    """Called when exception class RequestException is caught.
     The specific error is identified and the error log message is generated with a template.
     The specific metric is incremented.
     An HTTP error is returned.
     """
    error_type = type(e)
    for expected_error_type in expected_main_error_types:
        if error_type is expected_error_type:
            error_logging_metrics[expected_error_type.__name__].inc()
            logging.error(
                f"{error_logging_messages[expected_error_type.__name__]}" +
                f"type {error_type.__name__}"
            )
            return Response(
                status=error_logging_error_codes[expected_error_type.__name__]
            )
    error_logging_metrics["unknown error"].inc()
    logging.error(
        f"{error_logging_messages['unknown error']}" +
        f"type {error_type.__name__}"
    )
    return Response(status=error_logging_error_codes["unknown error"])
```

Listing 7: Logging message generation and emitting in views.py of the inventory microservice.

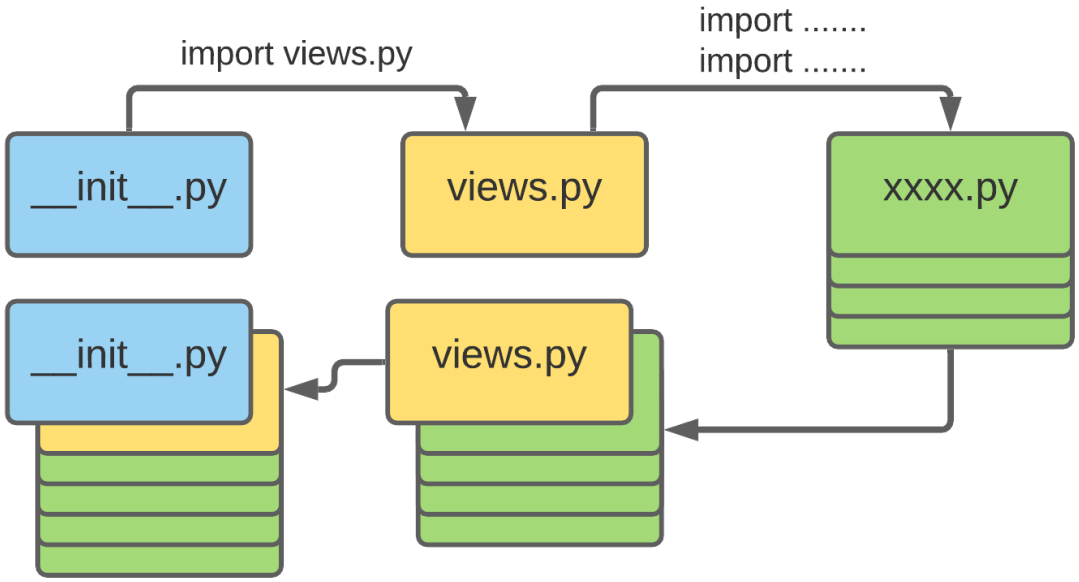

Figure 4.2: Illustration of Python's simplified import structure.

matted so that they contain meaningful information about the occurrence logged and its circumstances.

4.1.3 Monitoring and Metric Collection

Metrics are collected over the modules that comprise the microservices at two main points: the module in which web access is routed through functions (views.py), and the module which describes how events are formed and emitted (domain_events.py). The two main categories of these metrics are HTTP request and event counters, and exception counters. The former increases each time an HTTP request is received or an event is sent out, respectively, the latter increases when an exception occurs while the application is working. The exception counters are solely focused on exceptions raised by the requests package, as such, they are related to HTTP calls that the service makes to the EventStoreDB. Next to these, system resource metrics such as CPU and memory are also exposed; although, this is an in-built feature for Linux-based operating systems. Metrics are defined similarly in both services, as shown in Listing 8 and can be used in either one of the ways shown in Listing 9.

4.1.4 Observations

During the development process, problems were encountered relating to the methodologies used, as a single person was responsible for both microservices. This would not happen if a development team would work on a microservices-based application[4], and thus, this provides valuable practical experience in the practices set by the methodologies in use. Developers need to quickly realise the shared aspects of the services in development, so that those can be handled as dependencies, rather than duplicating code over a set of microservices. This is vital, as code duplication can lead to disparate changes in instances

[4]As detailed in Section 3.1, the microservices-based approach advocates for small teams that are responsible for a single service.

```
performance_metrics = {
    "http_request_counter": Counter(
        "inventory_http_request_counter",
        "Counter of HTTP requests being served",
        ["method", "endpoint"],
    ),
    "event_send_counter": Counter(
        "inventory_event_send_counter", "Counter of egress events"
    ),
    "network_timeout_error_counter": Counter(
        "inventory_network_timeout_error_counter",
        "Errors caused by timeouts while establishing connection",
    ),
    "http_error_counter": Counter(
        "inventory_http_error_counter",
        "Errors caused by HTTP unsuccessful status code response",
    ),
    "connection_timeout_error_counter": Counter(
        "inventory_connection_timeout_error_counter",
        "Errors caused by timeouts during data transmission",
    ),
    "redirect_error_counter": Counter(
        "inventory_redirect_error_counter",
        "Errors caused by exceeding redirection limits",
    ),
    "connection_error_counter": Counter(
        "inventory_connection_error_counter",
        "Errors caused by general network connection failure",
    ),
    "ambiguous_network_error_counter": Counter(
        "inventory_ambiguous_network_error_counter",
        "Errors caused by exceptions not specifically measured",
    ),
}

"""Initialise labels so they can be used later."""
performance_metrics["http_request_counter"].labels("GET", "/")
performance_metrics["http_request_counter"].labels("GET", "/health")
performance_metrics["http_request_counter"].labels("GET", "/metrics")
performance_metrics["http_request_counter"].labels("POST", "/create")
performance_metrics["http_request_counter"].labels("POST", "/delete")
performance_metrics["http_request_counter"].labels("POST", "/add")
performance_metrics["http_request_counter"].labels("POST", "/submit")
```

Listing 8: Prometheus metrics defined in prom_logs.py.

```python
@network_error_metric.count_exceptions(requests.exceptions.ConnectionError)
@http_error_metric.count_exceptions(requests.exceptions.HTTPError)
@timeout_error_metric.count_exceptions(requests.exceptions.Timeout)
@redirect_error_metric.count_exceptions(requests.exceptions.TooManyRedirects)
def execute(self):
    try:
        """Action resulting in an event being sent increments counter."""
        Event.event_counter_metric.inc()
    except:
        """In case of specific exception caught anywhere in this function,
       @ decorator increments counter.
       Attention: when multiple inheritance is possible it can cause multiple increments
       """
```

Listing 9: Two examples of the Prometheus Python client in use.

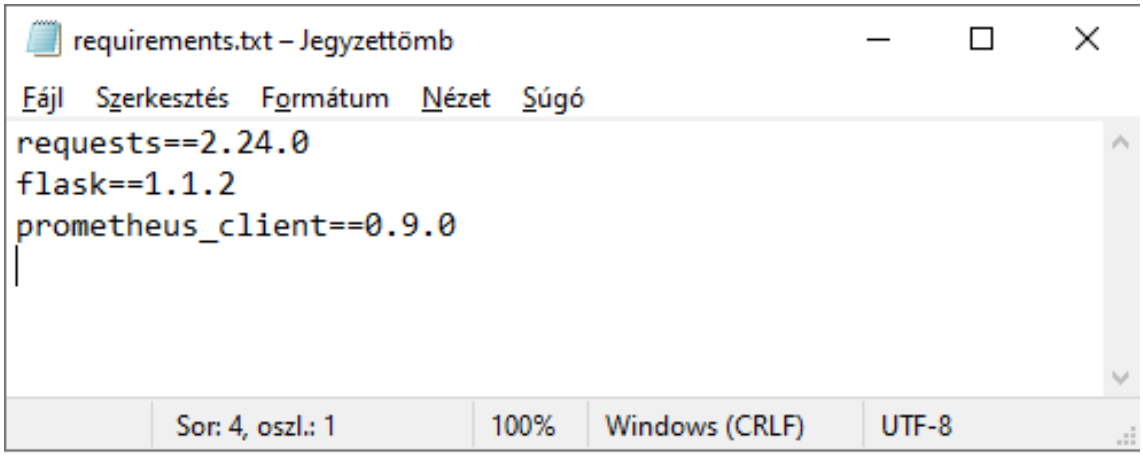

Figure 4.3: A list of required packages that are used by a microservice.

of the shared implementation. If this means that there would be problems in the cooperation of the microservices, it can cause application failure. The example of this is the implementation of events in either microservice. The design process did not consider how these aspects could be abstracted away from the codebase, so it remained seated inside each service's code. While the individual events are handled separately, as these are tightly coupled with the underlying business logic, the basis for these events is the same. This small portion is a duplication and should have been implemented separately.

There is also a strong correlation between the dependencies in use in both services. As this application is rudimentary, there is no need for extended injection of numerous dependencies. As can be seen in requirements.txt, a list of dependencies shown in Figure 4.3 used by Python's pip package handler, only three third-party packages are used. The best practice of defining exact versions of used third-party tools can also be noticed. However, by simultaneously developing two services, the development process had many parallels, similarities can be found between the implementations of the two services. Based on experience, the developer must pay close attention to the version of the packages in use and the capabilities they expect from it. If the configured version is different than the one referenced during development, incompatibility issues can occur. Conversely, the versioning

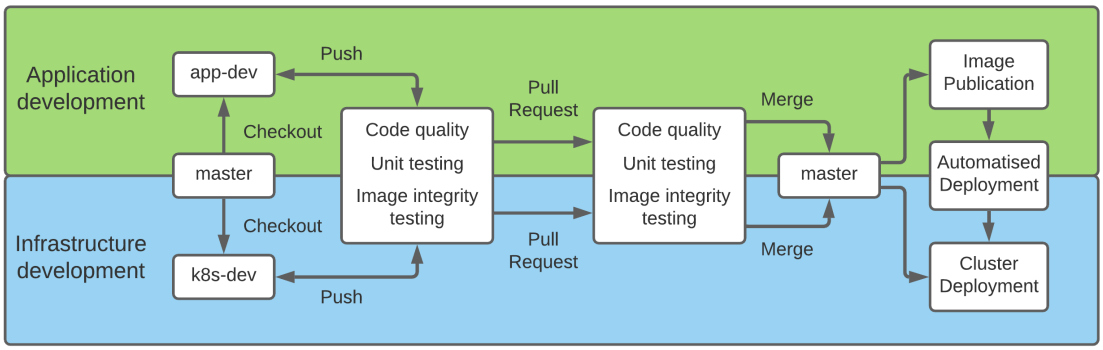

Figure 4.4: Visualisation of the CI/CDep processes used in the development process.

information in configurations surrounding the microservices has to follow the changes in the package versions developers might make over time. These are present in such places as the Python requirements list or the setup.py module, which defines the expected packages to be present when running a Python package. This is the job of the developer, as they are the ones using the dependency, developers and operators must be aware of each place a configuration information is stored or referenced. This shows a point of connection between developers and operators that methodologies aim to handle from Section 3.4, as the seamless operation of the developed software requires clear communication about the environment that the software needs to function in. This is an important part of DevOps culture, and the reason for the use of containerisation technologies.

4.2 Continuous Development Environment

The development process is supported by an automated environment that contains a cloud-hosted version control system, and automated CI/CDep pipelines. GitHub is utilised as the source code hosting platform, with Git as the local version control client. GitHub Actions is available through an API and facilitates the use of CI servers for scripted workflows handling processes such as code linting and image file publishing. The whole process that is to be detailed here is illustrated in Figure 4.4. Authentication against the cloud platform is performed with a kubeconfig file and secrets saved inside the repository. The source code is available at its GitHub repository [5].

The application and its Kubernetes infrastructure is stored in a monorepository, meaning that files of different processes are parts of the same version control process[5]. This can be seen in the repository's structure, shown in Figure 4.5.

The source code and configuration files of the two microservices are stored as Python packages under eventStoreTestApp, while the files defining the Kubernetes cluster housing these services can be found under k8s in separate folders based on function. The master branch always contains the version that is currently running in the cluster and reachable for its users, and also the declarative descriptions that are used to build the Kubernetes cluster running the application. This follows the best practice of keeping configuration files in a version controlled system, as it allows for a single source of truth about the current state of the cloud infrastructure, and, secondly, provides a single point where changes

[5]As opposed to a polyrepository structure, where different projects are stored in their own repositories, even if they closely interact with each other.

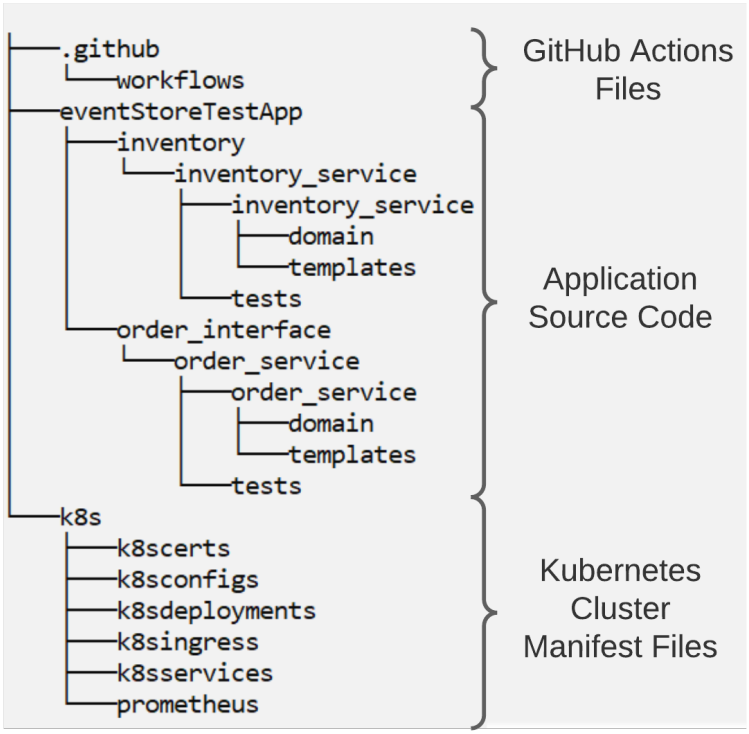

Figure 4.5: The folder structure of the application repository.

made are assured to be applied to every related system. However, the latter point involves much more related to CI/CDel/CDep processes.

Application and infrastructure development is carried out by branching out from this master branch into two branches, app-dev and k8s-dev, respectively. This follows guidelines such as GitFlow [51], which give special meaning to each branch used, and a specific structure for branching out, defining which one can be merged together with which, for example. The process used in this thesis is a simplified example, shown in Figure 4.6.

While working on these branches, code quality, unit, and image integrity tests are performed. Each push that the central, online repository receives from a local repository in place on a developer's machine runs these processes, with the exception of infrastructure development, as it does not produce image files or run unit tests. The purpose of these processes are to provide feedback about the compliance of a developer's code to centrally defined rules and best practices. For example, Python and YAML code pushed must comply to syntax rules defined in the Black (Python-based) and YAML linter, which are used for checking code quality. These linters check the code pushed each time. Image integrity tests make sure that the pushed code is able to be containerised and run as an image. Unit tests are defined in each microservice's source code, and the pytest framework [29] is used each time to run those test and output its feedback. Testing has not been included in this thesis, only the framework has been integrated. However, in agile methodologies, testing plays an important part in everyday development processes.

If all checks pass, and the developer wants to integrate changes into the running application or infrastructure, they need to make a pull request to the master branch, starting a new set of processes. Firstly, the same processes run as before, but this time the changes compared are the whole new branch aggregated and the currently running instance on the master branch. If these checks pass, the pull request can be merged into the master

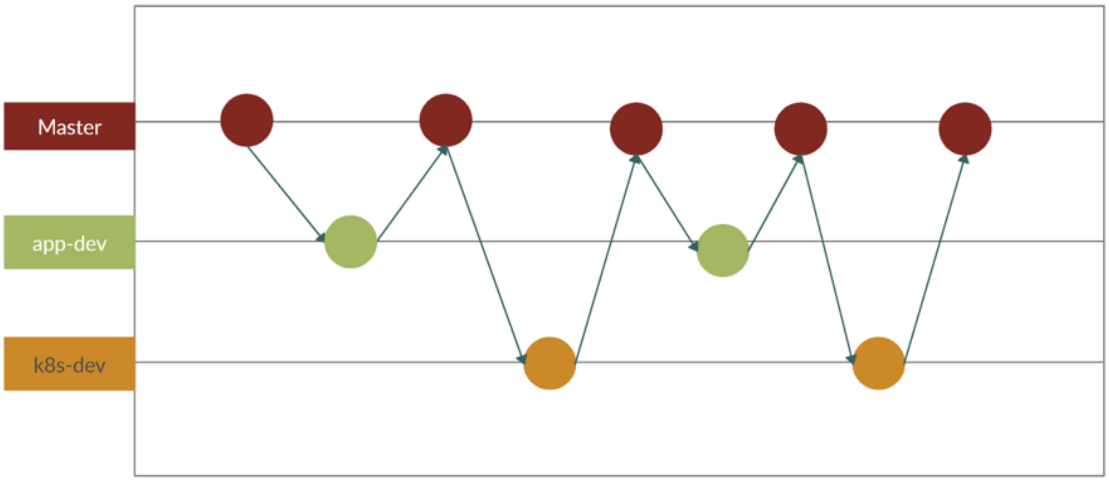

Figure 4.6: Visualisation of the git workflow in use during application and infrastructure development.

branch. Here, based on what kind of changes have been made, two things can happen. If changes are situated inside the eventStoreTestApp, the image publisher pipeline makes a Docker image and publishes it to the GitHub Container Registry, which is connected to the repository in use. This way the new version of the application takes the place of the old on the master branch and the container registry as well, with a new version tag. If changes were made inside the k8s folder, then a process runs which authenticates against the cloud platform in use to run the Kubernetes cluster, and the kubectl utility is used to apply the updated descriptor files, causing changes where the declarations have changed.

At this point, branches have been merged into the master, and no further action is needed on the part of the developer or operator; however, best practice dictates that versions must be added to any configuration that contains third-party software. This version is changing each time a new application version is released, and if new versions should immediately be deployed on release[6], manually changing these numbers in the Kubernetes descriptor files is not efficient. For this, another process has been defined, shown in Figure 4.7. Kustomize [43] is a tool which allows dynamic configuring of YAML templates. The practical example in this case is that each time a new version is released in the container registry, the version number is written inside a Kustomize file, which is in turn used while applying changes through kubectl. What happens here, is that the actual version is not defined inside the YAML file that is used to deploy the application to the cloud cluster, it is automatically appended by the Kustomize tool. Thus, if Kustomize is used with kubectl and cloud provider authentication, a string of processes can be made, where simply changing the version number to the newly released one and using that Kustomize file while applying changes to the cluster with kubectl, the deployment of the new release is automated. Practically, this happens when the GitHub Actions release workflow triggers an automated change, which clones the repository, changes the version number, pushes the change back to the repository that called this workflow in the first place, and triggers the Kubernetes deployment workflow. This deployment will then use the new version number, inducing a change in the deployed application instance inside the cluster[7].

This set of procedures was influenced by continuous practices and GitOps. CI servers are configured using the GitHub Actions API by files contained under .github/workflows

[6] As is the case with Continuous Deployment.
[7] Actually, this workflow is the same that runs when files inside the k8s folder change.

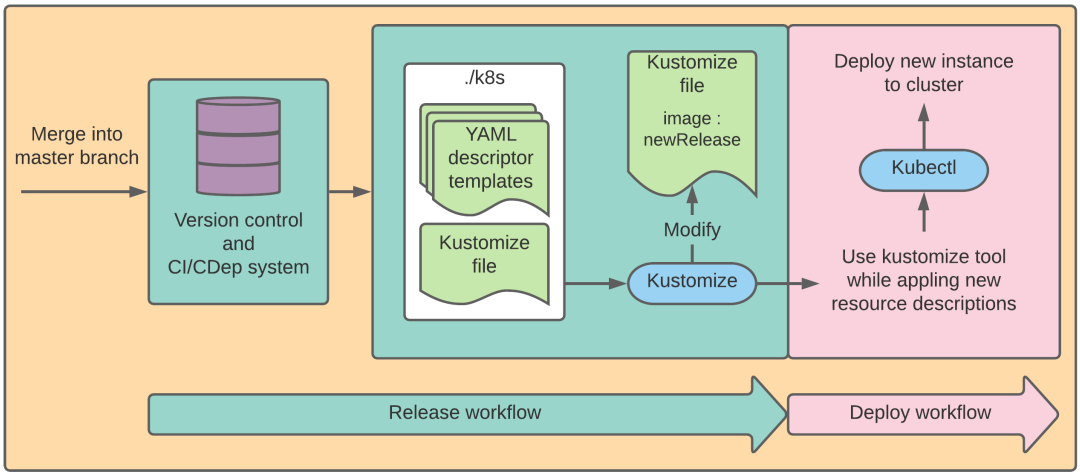

Figure 4.7: Visualisation of the release process using Kustomize to automate application versioning.

inside the repository. Each time one of the previously described triggers occur, a separate server is started where the configured actions are performed. This ensures, that the developer only needs to interact with the whole system through the version control tool. As described in Section 3.4.2, continuous deployment is the practice that integrates all the other practices, continuous integration and continuous delivery as well. By considering, that from development to deployment everything is automated, this thesis' continuous development environment implements the whole continuous ecosystem. The description of the running infrastructure is contained inside files under k8s, and everything is versioned down to the version numbers used in deployment, conforming to GitOps principles.

4.3 Cloud Cluster Operation

The Kubernetes environment that is used to host the application is maintained by multiple operators, not just this writer. Most of the deployment tasks are handled by processes described in this thesis. These include the deployment of the application, and most of everything described in Section 2.3.2. Other functions are handled by other operators, though, such as the deployment of different Kubernetes Operators, mentioned at the end of Section 2.3.2. The scope of this thesis only includes those functions that are maintained by this writer, other functions will only be mentioned as necessary.

A Kubernetes cloud environment was used to host the application developed in this thesis. Next to the processes that involved deploying and updating the elements of this application, there were a number of operations support tasks that were required related to monitoring, such as log and metrics collection, and alerting. These tasks and the Kubernetes cluster environment itself is outside of the scope of this thesis. The following description heavily relies on the previous description of a standard Kubernetes cluster in Section 2.3. As that correlates with the structure implemented in this section, specifics can be compared to the overview provided here.

4.3.1 Cloud Platform

The cloud cluster hosting this thesis' application is based on the Amazon Web Services cloud platform. This platform is entirely handled by third-parties, operative access is given only to the cluster that is preconfigured for use. This means that any cloud computing resource that the cluster has available, mainly CPU cores and memory, is outside of the responsibility of this thesis, operative tasks are focused on cluster management. Inside the cloud platform, a Kubernetes cluster has been installed and access given to it through a kubeconfig file. This file contains cluster and authentication information that can be used to gain access to a specific cluster's API server. This file is used to manually access the cluster, and by processes defined in Section 4.2 when automatically applying changes made during the development process, each by providing this file to the kubectl tool.

4.3.2 Application Deployments

To deploy the application, Deployment objects are used. As described, these objects use ReplicaSets to deploy similar Pods of the same specifications. The Deployment object is used to ensure the scalable management of these ReplicaSets. This means that when a configuration change is applied, the control process that governs the cluster's operation will replace these Pods in graceful manner, according to rules defined in the Deployment object description. Specifics can be found in [39], generally, the process will ensure that new Pods are replaced gradually instead of tearing the old Pods down at once, ensuring smooth transition. Listing 10 shows how to define the Deployment object for the inventory service.

Three of these are used inside the cluster, one Deployment for each of the two services - inventory and order services - and the EventStoreDB node. The two service contain more specifications than the EventStoreDB, as these require more configuration to work. What is similar is the way the containers receive the images they need to run and the specifications of the ReplicaSets. The image names and versions are set through the Kustomize tool, already presented in Figure 4.7, which are then downloaded automatically from the GitHub Image Repository or other image repositories. The number of Pods to run is given in the spec.replicas value, and every other configuration to be used while deploying the Pods is described further in the ReplicaSet specification.

Two aspects specific to the deployment of the two services are the liveness probes and the config volumes. Liveness probes are carried out by the Kubernetes control process to the defined HTTP port to monitor the functioning of the container. A specific path is implemented for this, returning 200 status if successful[8]. The config volume is provided so that environment-specific variables can be injected into the application running in these Pods. The network location of the EventStoreDB node is needed so that the application can communicate with it. However, in order for the system to be scalable, and a best practice in microservices architecture, the exact location cannot be hardwired into the application code, it has to be provided by the environment. A ConfigMap object, the one shown in Listing 11, is used to define this value. Thus, this value can be parsed as an environment variable according to the Pod template. This way, when a Pod is initialised and a container starts running, its environment variables will already contain the necessary values for it to function. The exact value is the DNS name of the Service that handles access to the EventStoreDB Pod.

[8]It was not implemented in this thesis; however, the EventStoreDB also has this function. Thus, it also uses a liveness probe.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inventory-deployment
  labels:
    app: inventorymicroservice
spec:
  replicas: 2
  selector:
    matchLabels:
      app: inventorymicroservice
  template:
    metadata:
      labels:
        app: inventorymicroservice
    spec:
      volumes:
        - name: config-volume
          configMap:
            name: eventsourcing-testapp-configmap
      containers:
      - name: inventory
        image: ghcr.io/bproforigoss/inventorymicroservice
        env:
          #Variables are imported from ConfigMap here
          #Excluded for space saving
          - name: EVENTSTORE_STREAM_URL
            #Previously imported variables referenced here
            value: >
              "$(EVENTSTORE_WEBSCHEME)://"
              "$(EVENTSTORE_URL):"
              "$(EVENTSTORE_STREAM_PORT)/streams"
        volumeMounts:
          - name: config-volume
            mountPath: "/etc/config"
        livenessProbe:
          httpGet:
            port: 5000
            path: /health
        ports:
        - containerPort: 5000
          name: http
```

Listing 10: Deployment YAML manifest for the inventory service.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eventsourcing-testapp-configmap
data:
  eventstoredb_webscheme: "http"
  eventstoredb_location: "eventstoredb-service"
  eventstoredb_stream_port: "2113"
```

Listing 11: YAML manifest for the ConfigMap object.

4.3.3 Routing Inside and Outside the Cluster

In order for these Pods to be easily reachable, separate Services are configured for each. These have DNS entries that do not change, even with the reconfiguration of the Service object itself. Thus, their location can be statically set in other objects by using their DNS entry name. They provide load balancing and routing to the Pods. The load balancing aspect is evident when looking at Figure 4.8. It shows the order service under a load testing. The two graphs represent either one of the deployed Pods and the number of events they send, caused by the requests coming in at the path "/create". The two graphs rise steadily, showing that the two Pods share the load equally. This is not enough for accessibility on the public internet, though. For this, an Ingress Controller has been integrated. This is one of the controllers mentioned in Section 2.3.2, providing a standard API definition of a service that needs management processes not native to Kubernetes. An Nginx HTTP proxy is used to provide routing to the internet through three domains: db.bprof.gesz.dev, order.bprof.gesz.dev, and inventory.bprof.gesz.dev, routing for the database, the order service, and the inventory service, respectively. The exact configuration of this proxy and its controller is handled by third-parties, only the routing rules are configured in this thesis. The general method used here is to take the domain name and separate it from the rest of the URI, this takes up the path that can be matched with the path that is defined in relation to functions of the application, mentioned in Section 4.1.1. Once the path element is known, the ingress proxy can route the traffic to the Service element defined in relation to the URI used to reach it. For example, traffic to inventory.bprof.gesz.dev will be routed to the Service of the inventory Deployment. This configuration is defined as shown in Listing 12.

4.3.4 Cluster Certificates

Exposing the two services to the public internet from the cluster would not be possible without SSL certificates, as HTTPS is required to reach the cluster. In order for these certificates to be available, another Operator, a Certificate Manager is needed. The Certificate Manager ensures that the SSL certificates used by the cluster are operable, that they are requested and issued correctly, and that they remain up-to-date. For this, Let's Encrypt's certificate services are used. Like the Ingress Controller, the Certificate Manager is a portion of the cluster infrastructure that is not maintained by this writer, it is outside of this thesis' scope. With a process for certificate requisition available, a Certificate object needs to be defined for the actual domains that are to be validated. This object simply defines the domain names that are used by the cluster, the aforementioned three ending in bprof.gesz.dev, that need to be validated. These domains can then be used

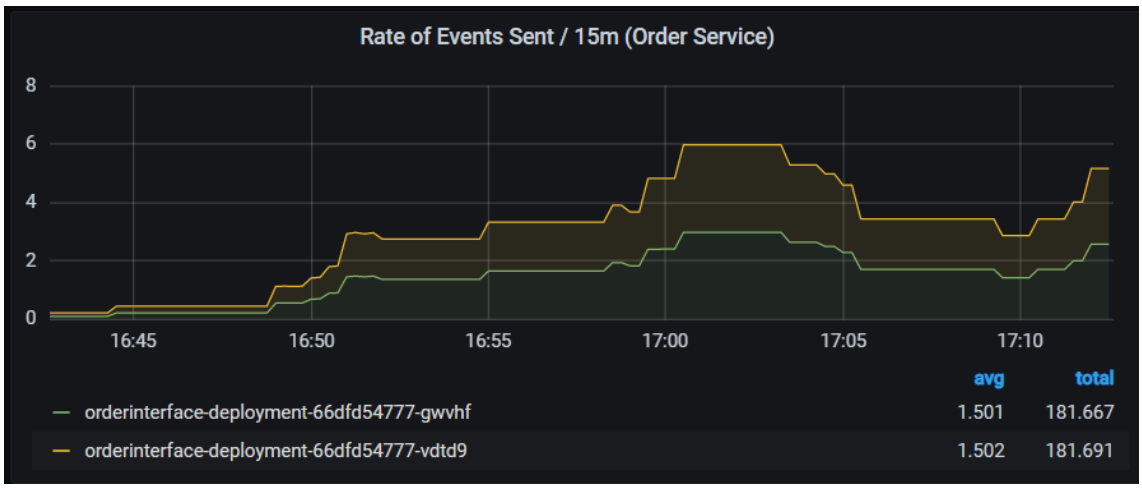

Figure 4.8: Grafana panel showing the events emitted by the order service under load testing.

```yaml
rules:
    - host: inventory.bprof.gesz.dev
      http:
        paths:
        - path: /(.*)
          pathType: Prefix
          backend:
            service:
              name: inventorymicroservice-service
              port:
                name: http
    - host: order.bprof.gesz.dev
      http:
        paths:
        - path: /(.*)
          pathType: Prefix
          backend:
            service:
              name: ordermicroservice-service
              port:
                name: http
    - host: db.bprof.gesz.dev
      http:
        paths:
        - path: /(.*)
          pathType: Prefix
          backend:
            service:
              name: eventstoredb-service
              port:
                name: admin
```

Listing 12: Routing rules defined in the Ingress YAML manifest.

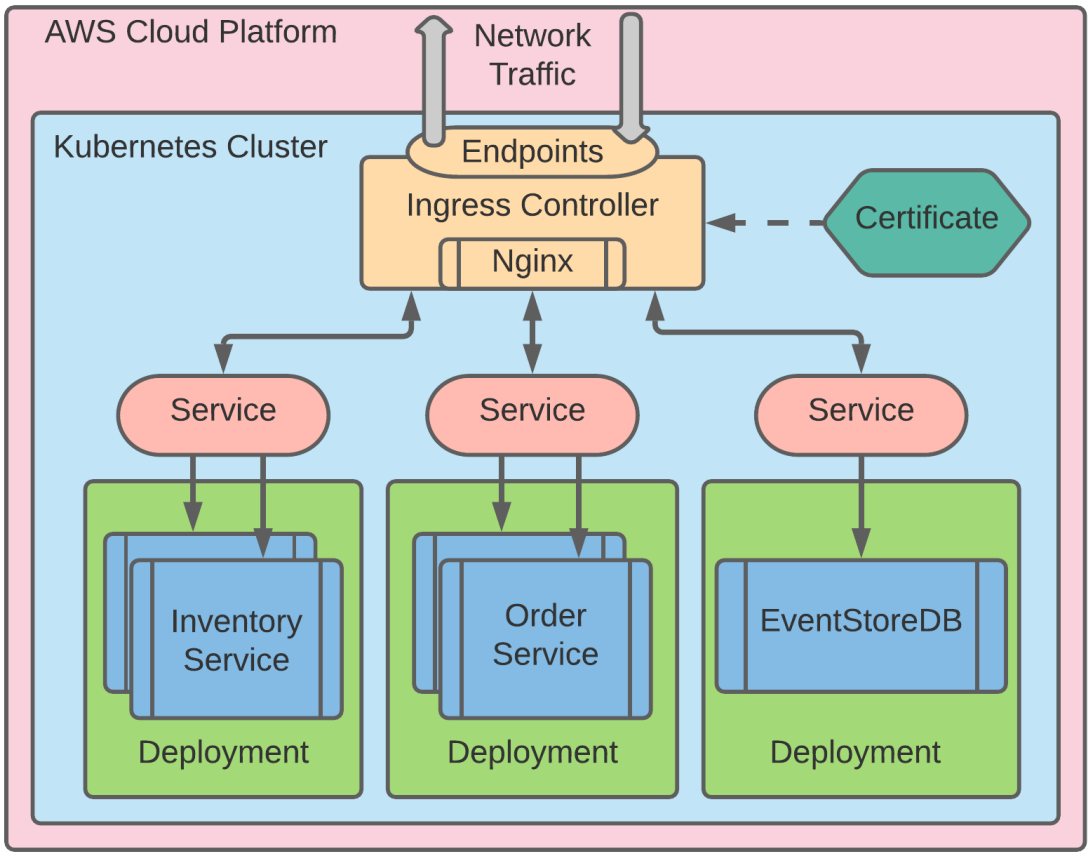

Figure 4.9: Simple visualisation of the cluster environment maintained in this thesis.

in other objects to expose services, as authentication can be provided using the issued SSL certificate.

These parts make up the basic fabric of the cluster that the application operates in. Logging and monitoring has not been included, as the specifics will be detailed in another section. The cluster that has been discussed is visualised in Figure 4.9.

4.4 Cloud-Native Monitoring, Logging and Alerting

In Section 3.3.2, additions to the set of practices called the Twelve-Factor Application has been detailed, and the addition of telemetry mentioned. The telemetry factor considers that applications deployed over numerous servers, possibly across the globe, can get out of reach of the developers responsible for them. Access may not be possible, or may be restricted, such as in the case of a strong security policy separating developers from deployment environments. This necessitates the use of indicators about the health of the application. In this thesis' case, metrics and logs have been implemented inside the developed services to properly monitor them. The solutions used to perform this monitoring are detailed here, in as much as they are operated inside the scope of this thesis.

```
from prometheus_client import Counter
"""Prometheus client must be installed separately."""

"""This is a Counter type of metric, it only increments.
   First argument names the metric examplecount.
   Second argument gives a detailed explanation supplied with the metric as well.
   Third argument defines what kind of labels can be added to the metric.
   """
example_metric = Counter("examplecount",
                         "An example counter",
                         ["country", "city"])

"""The metric is incremented by one by a specific set of labels."""
example_metric.labels(country="Hungary", city="Budapest").inc()
```

Listing 13: Example implementation of a Prometheus metric in Python.

4.4.1 Metrics

Metrics are collected and managed by Prometheus, a metric management tool. "Prometheus fundamentally stores all data as time series: streams of timestamped values belonging to the same metric and the same set of labelled dimensions" [24]. Data are stored as a time series, all metrics exposed through an HTTP endpoint scraped by Prometheus are read - or "scraped" - between set time intervals. These intervals provide individual points of time at which the metrics' state is known. From this, a series can be made, showing the progression of a metric's state by the interval set. For example, if a scrape time is set at 10 seconds, every 10 seconds Prometheus initiates a scrape of the metrics, and data points may be read at a minimum of 10 seconds intervals, if successful. A simple example can be seen in Listing 13. Here, labels are used as well. As quoted, these provide further individualisation of metrics of the same name. Prometheus, by default, uses the pull method to collect these metrics, meaning it initiates the read itself instead of waiting for the applications to push their values individually. Once gained, metrics are managed and stored by Prometheus, they are queryable as graphs or tabular data with a specific query language, PromQL [23]. The endpoints that can be accessed to read Prometheus metrics are configured in Prometheus' configuration files as hosts and ports (such as localhost:8080), these are expected to expose a "/metrics" endpoint. This level of detail is sufficient for the functions in use inside this thesis' scope, further details about Prometheus can be found in [25].

In this thesis, both services expose metrics through the "/metrics" endpoint, detailed in Section 4.1.3. An example of how these metrics are exposed as formatted Prometheus metrics, and how this is programmed can be seen in Listings 14 and 15.

As already mentioned, Prometheus needs to have the host and port values configured to know what destination it needs to scrape. However, while operating inside a Kubernetes cluster, IP addresses can change frequently as Pods are scheduled and deleted. This requires the frequent reconfiguration of Prometheus, which is inefficient if done by hand. Prometheus is a tool that needs an Operator to work in a cloud-native environment, the Prometheus Operator [27]. This Operator is outside of the scope of this thesis; although, there is a task that has to be performed for it to function. The ServiceMonitor declares how

```
# HELP inventory_http_request_counter_total Counter of HTTP requests being served
# TYPE inventory_http_request_counter_total counter
inventory_http_request_counter_total{endpoint="/",method="GET"} 6.0
inventory_http_request_counter_total{endpoint="/health",method="GET"} 44363.0
inventory_http_request_counter_total{endpoint="/metrics",method="GET"} 14788.0
inventory_http_request_counter_total{endpoint="/create",method="POST"} 0.0
inventory_http_request_counter_total{endpoint="/delete",method="POST"} 0.0
inventory_http_request_counter_total{endpoint="/add",method="POST"} 0.0
inventory_http_request_counter_total{endpoint="/subtract",method="POST"} 0.0

# HELP inventory_event_send_counter_total Counter of egress events
# TYPE inventory_event_send_counter_total counter
inventory_event_send_counter_total 0.0

# HELP inventory_timeout_error_counter_total Errors caused by request timeouts
# TYPE inventory_timeout_error_counter_total counter
inventory_timeout_error_counter_total 0.0

# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 2.8114944e+07

# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 147.16
```

Listing 14: Excerpt from metrics exposed by the inventory service on "/metrics".

```python
@app.route("/metrics", methods=["GET"])
def metrics():
    """Requests to this endpoint are also measured."""
    http_counter_metric.labels(method="GET", endpoint="/metrics").inc()
    readings = []
    """The in-built metric collector object is read for the latest
     state of its metrics.
     """
    for metric in prom_logs.performance_metrics.values():
        readings.append(prometheus_client.generate_latest(metric))
    """A separate metric collector for system resources (CPU, memory) is used."""
    readings.append(
        prometheus_client.generate_latest(prometheus_client.PROCESS_COLLECTOR)
    )
    """All the metrics and their states are returned as plain text."""
    return Response(readings, mimetype="text/plain")
```

Listing 15: Exposition of Prometheus metrics inside the inventory service.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: inventorymicroservice-servicemonitor
  labels:
    component: microservice
spec:
  selector:
    matchLabels:
      app: inventorymicroservice
  endpoints:
    - port: http
```

Listing 16: ServiceMonitor YAML manifest for the inventory service.

Kubernetes Services should be monitored, keeps the Operator up-to-date on the current state of the Services, and through them the Pods, in the API server. The Operator keeps Prometheus' scrape configuration current according to that state. An example of the ServiceMonitor object is shown in Listing 16. These ServiceMonitors have two main values: a selector label and a port name. The selector label is used to find all the Pods that Prometheus has to scrape, comparing the labels that each Pod has to the label set for the ServiceMonitor. In Listing 16, this is "app: inventorymicroservice". The port name identifies the port that can be used to make the host and port combination that is finally configured for Prometheus. In Listing 16, this is "- port: http", a named port whose name has a relation to a port's name in the inventory services' Pod definition. The manner in which this object connects Pods to a Prometheus instance is illustrated in Figure 4.10 [26].

4.4.2 Logging

Metrics are good for measuring quantifiable performance data; however, they do not provide as much information as logs can. While metric numbers quantify software performance, logs provide detailed information about events that transpire while the application is functioning. Logging has been implemented as detailed in Section 4.1.2. The logs emitted by the services are routed to the standard output, which is where Promtail scrapes them.

Promtail is an agent that collects logs from running applications and carries them over to a log collection solution. There are different sources Promtail can tail logs from, in this application, this is the standard output. Every location scraped has a Promtail daemon running, multiple instances have no knowledge of each other [28].

Loki is the log aggregation solution used to store and manage these logs. It saves the logs by indexing their metadata, with the actual logs being saved in compressed object chunks. This way, Loki saves cost and simplifies operation [19]. When log messages are formatted as key-value pairs, Loki parses these fields and outputs them as such. Figure 4.11 shows how these logs can be inspected inside Grafana when Loki is the data source. These solutions are operated outside the scope of this thesis and additional configuration is not needed on the part of the application operator for them to be functional.

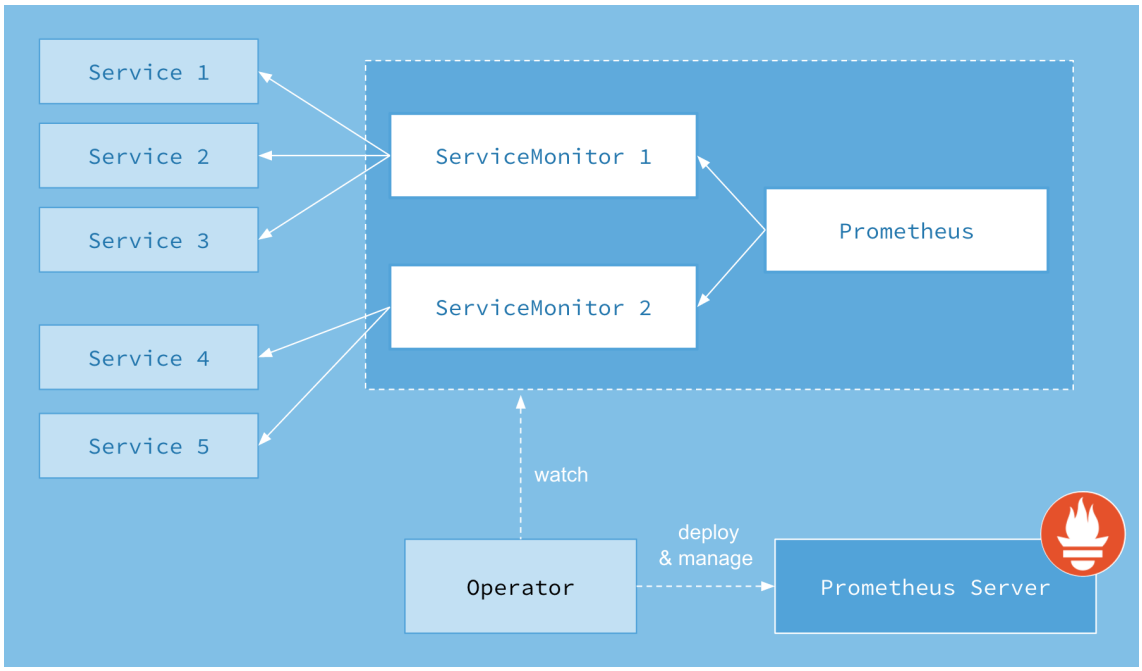

Figure 4.10: Illustration of the ServiceMonitor object in use to connect applications to the Prometheus instance.

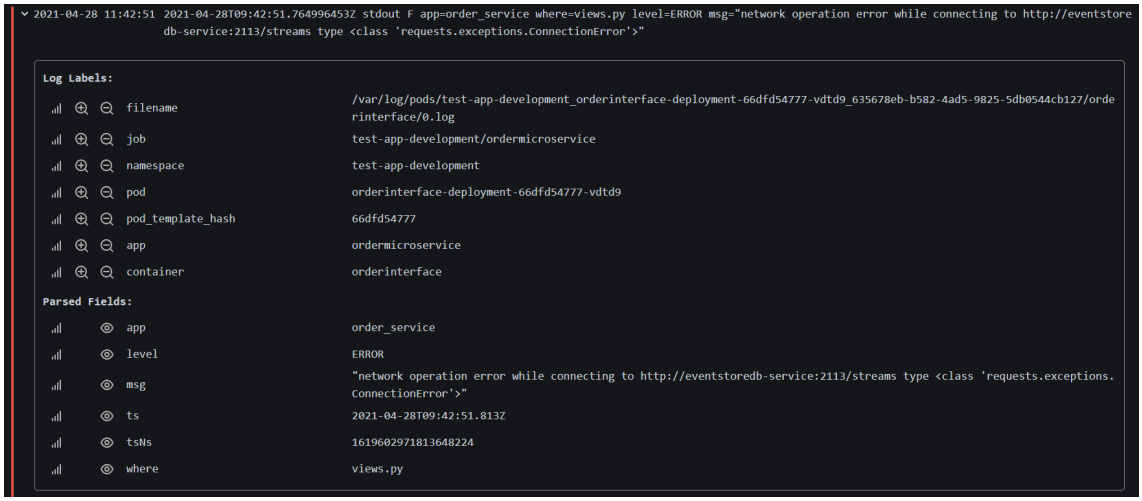

Figure 4.11: Error log collected by Loki in Grafana's explorer mode.

4.4.3 Monitoring

The data collected by the aforementioned solutions and mechanisms built into the services provide good information on the health of the services. However, they are not easily digestible. Prometheus has limited visualisation capabilities, and the logs stored by Loki are not categorised further than the data they are indexed by. In order to visualise the data, Grafana is used.

Grafana is a data visualisation tool where dashboards can be created with panels showing data from variable data sources in different kinds of representation. This is one of the main strengths of Grafana, dashboards can be easily customised with panels. These panels can show graphs, gauges, tables, histograms and heat maps. This set of representative tools can further be expanded by community plug-ins. It has a dynamic way of outputing data, as the queries that make up the data included in a panel can be set as variables, instead of being static. An example for a dashboard can be seen in Figure 4.12, while how multiple queries and variables are used can be seen in Figure 4.13. Another strength of Grafana is that it can handle multiple data sources. In this case, these are Prometheus for metrics, and Loki for logs. However, many other data sources can be included, and this allows for holistic representation of the whole system in use. Utilising all these data, alerting can be set up as well, where, when preset conditions are met[9], integrated alerting solutions can be used to push messages to operators and inform them about steps that need to be taken. These tools include a plug-in for the Slack communication platform, and alerting tools such as PagerDuty [17].

For this system, Alertmanager [4] has been deployed next to Prometheus. Alerts can be defined with data about the specifics of the alert and the condition that must be met inside Prometheus. When the condition is met and remains true for a specific amount of time, Prometheus fires an alert that Alertmanager catches. Handling smaller tasks, such as grouping and deduplicating, Alertmanager also routes the alert to the receivers that have been integrated, Slack in this case. This way, operators can use different clients to keep up-to-date on the condition of their systems without the need for constantly monitoring their state through the tools mentioned in this section.

The Grafana instance and the Alertmanager are handled outside of this thesis' scope, operations tasks connected with it are the use of features provided by it. Apart from the mentioned system resource monitoring example in Figure 4.12, panels have been created that monitor HTTP requests, egress events from services that use network bandwidth, and errors and their types. These are shown in Figures 4.14 and 4.15. A specific alert has also been defined as shown in Listing 17. This alert fires when the combined rate of network errors originating from the serving of HTTP requests received divided by all HTTP requests received raises above 5% and remains there for at least 10 minutes.

[9]For example, some utilisation statistic remains above a set level for a set amount of time.

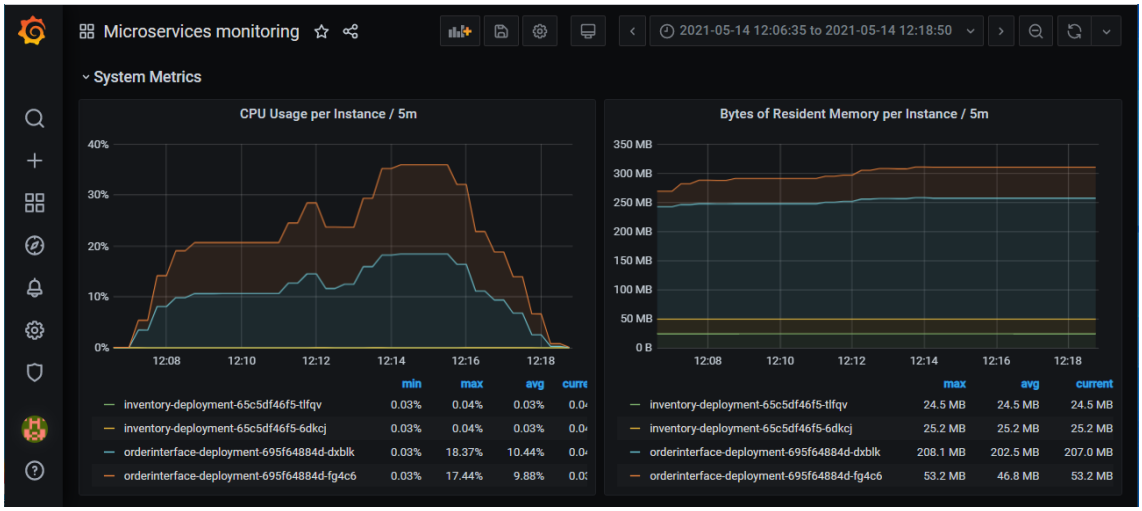

Figure 4.12: Grafana dashboard showing system resource data of the deployed services.

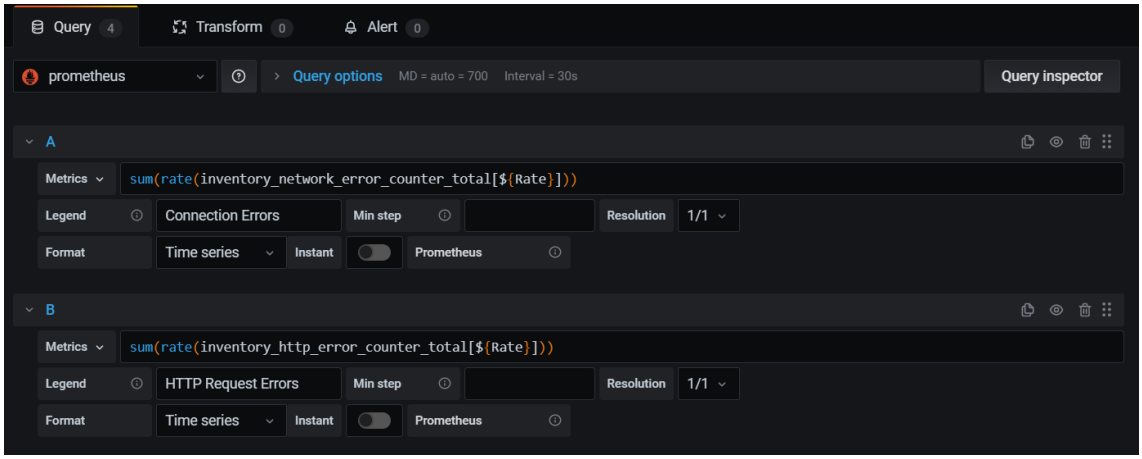

Figure 4.13: Multiple queries used in a single Grafana panel.

Figure 4.14: Performance metric panels measured from the deployed services.

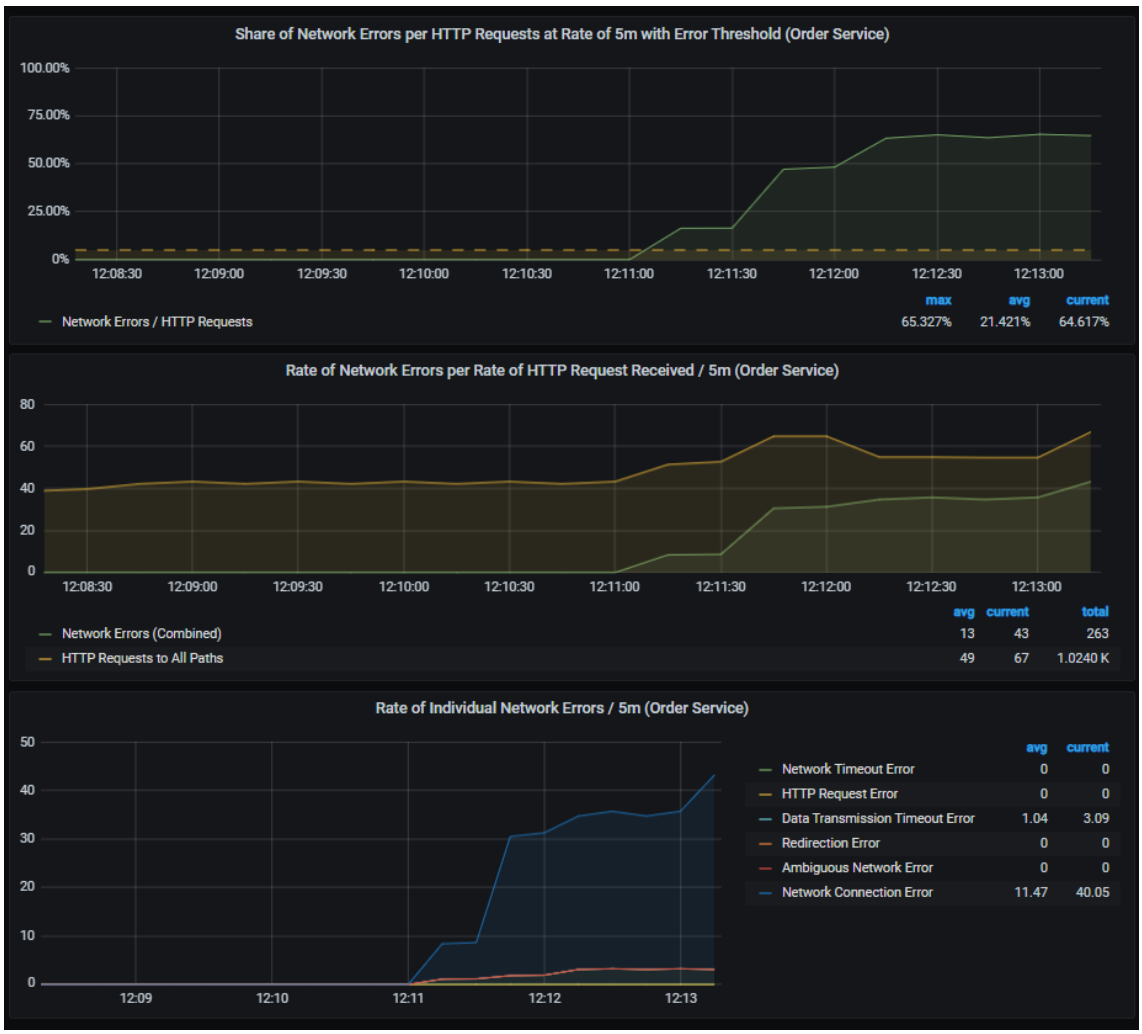

Figure 4.15: Error metric panels measured from the deployed services.

```yaml
- name: orderservice.rules
  rules:
    - alert: OrderServiceConsistentNetworkErrors
      annotations:
        description: >
          The rate of network errors compared to HTTP requests
          has been above 5% for 10 minutes.
        summary: Order service is constantly emitting network errors.
      expr: >
        sum(rate(order_network_timeout_error_counter_total[5m]) +
        rate(order_http_error_counter_total[5m]) +
        rate(order_connection_timeout_error_counter_total[5m]) +
        rate(order_redirect_error_counter_total[5m]) +
        rate(order_connection_error_counter_total[5m]) +
        rate(order_ambiguous_network_error_counter_total[5m]))
        / sum(rate(order_http_request_counter_total[5m]))
        > 0.05
      for: 10m
      labels:
        severity: warning
```

Listing 17: Definition of an alert inside Prometheus connected to the order service.

```
hey -z 60m -c 200 -q 200 -m POST https://order.bprof.gesz.dev/create
```

Listing 18: The script used to load test the cloud system using the hey tool.

4.5 Use Case of Cloud-Native Software Operation

In order to prove that the system described in previous sections is operable and works as expected, a load testing scenario will be detailed in this section. Necessarily, some preexisting conditions were deliberately injected into the system that was tested for it to perform as needed; however, these conditions were designed to be realistic. The setup to be detailed could be present in any production environment, taking the scale of the resources and the services into consideration.

The load testing scenario was run using the hey HTTP load generator [36]. This tool was used to constantly create HTTP requests at a set rate in order for the load to be balanced and the test to be reproducible. Different Kubernetes resource limits and load distributions were tested before the scenario. The goal was to produce a load that caused a realistic amount of errors stemming from the application overloading the database with HTTP requests[10]. This would, over time, produce an alert and initiate the process of mitigating the effects of the error, and through development processes, finding the root of the problem and solving it. It was found that using the load described in Listing 18 and limiting the resources of the database according to Listing 19 was a good combination for the error rate to fluctuate constantly around 10% of all HTTP requests received by the application.

These configurations mean firstly that POST requests will be sent continuously for 60 minutes (-z handle) to the defined URL at a per second rate limit of 200 (-q handle) and with a limit of 200 concurrently running requests (-c handle). This generates a steady flow of requests independent of the hardware environment the test is run in and the limit on concurrency enables the all-time load to be limited to a set level. Secondly, the limit on the CPU resources of the database represents a real aspect of cloud operation, as resources are not unlimited, only available on-demand. This means that a steady limit has to be set on how much resources a cloud system element can use in order to keep operative costs predictable. This limit is subject to projections on predictable user load, and these projections could be faulty. This is the main preconception of this load test, that the initial resource allocation was underestimated. The configuration in Listing 19 means that every container inside Pods managed by the Deployment responsible for the EventStoreDB (initially one in this case) will be limited to a maximum of 100 millicores of a CPU, or 0.1th fraction of a virtual CPU per Core [7]. Limit is a specific term in Kubernetes resource allocation, it means that this resource shall never exceed this amount[11].

Thus, the injected changes into the system are the following: a resource limit on the CPU utilisation of the database Pod, and a previously not defined change in the codebase. This change represents a development oversight that impedes the performance of the application itself. The requests module allows for setting the timeout value on an HTTP request made by the client. The developers set this value inside the function responsible

[10] As described, the system is distributed and its elements communicate over HTTP.

[11] Another configuration is request, which means that the resource requires the set amount of the resource; however, it can exceed this amount, if possible.

```yaml
# Inside the eventstore-deployment Deployment manifest
spec:
  # ReplicaSet specification
  template:
  # Pod template specification
    spec:
    # Pod specification
      containers:
        - name: eventstoredb
          image: eventstore/eventstore
          resources:
            limits:
              cpu: "100m"
```

Listing 19: Excerpt from the eventstore-deployment Deployment manifest using CPU resource limits.

```python
def execute(self):
    Event.event_counter_metric.inc()
    es_id = uuid.uuid4()
    headers = {
        "Content-Type": "application/json",
        "ES-EventType": self.event_type.value,
        "ES-EventID": str(es_id),
    }
    requests.post(
        f"{os.getenv('EVENTSTORE_STREAM_URL')}/{self.aggregate_id}",
        data=json.dumps(self.data),
        headers=headers,
        timeout=1,
    )
```

Listing 20: Excerpt from the order service's domain_events.py module with its initial timeout value.

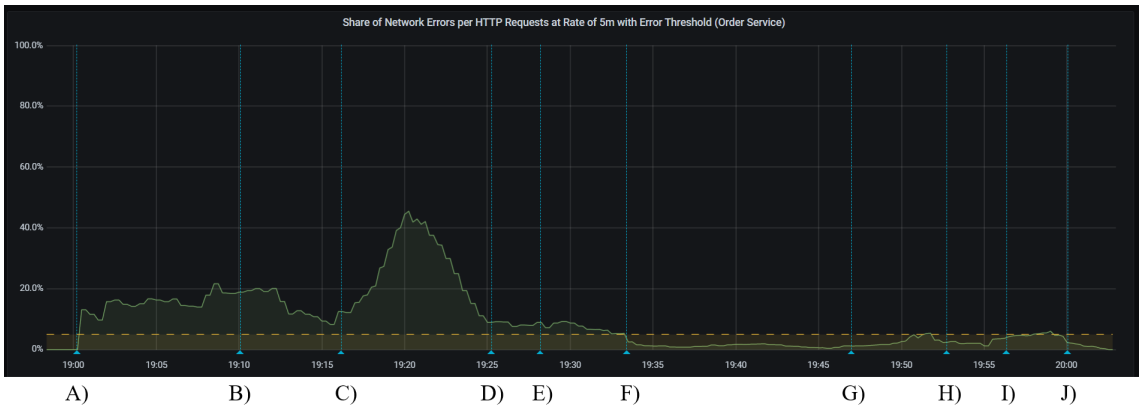

Figure 4.16: Error metric panel measuring the values used for alerting with annotations of specific events during load testing.

for sending events from the order service to the event store to a minimal value, one second, as shown in Listing 20.

With this setup, development and operations errors are present inside the system. This provides insight into the processes that surround cloud-native development and operation, the same processes that have been implemented in this thesis.

4.5.1 Load Testing Scenario

This section will detail the load testing and its findings. A main point of reference will be the specific Grafana panel that visualises the metrics used for defining the alert that is expected to fire during the scenario. Annotations have been made on the panel at set times, these will be referred to by their capitalised letter. This panel is shown in Figure 4.16.

At 19:00 the load testing was begun with the hey script described previously, annotated as event A). The generated traffic quickly raised the error rate above 5% and fluctuated between 9% and 22% when rated at a rate of 5 minutes[12]. This may go unnoticed by operators; however, there is an alert defined that constantly monitors the same expression that is used to calculate the value shown in the panel. As seen in the figure, this error rate never drops below 5%, and after 10 minutes, an alert fires, at annotation B). This alert is caught by the Alertmanager and sent to Slack through an integration that allows messaging on alert. This alert message is shown in Figure 4.17.

This is the moment an operator notices the high rate of errors. It is seen that there is an increasing load on the system and horizontal scaling is used to more evenly distribute the load on the database. At annotation C) the replica number of the EventStoreDB Deployment is raised from 1 to 2 manually to allow time for a permanent solution. This is done in the way shown in Figure 4.18 by editing the object description inside the system manually through kubectl. Its effect is shown in Figure 4.19, notice the difference between the ages of the two Pods. This shows that the previous Pod was not dismantled, instead, a

[12]The metrics used are counters, meaning they only increase. The rate function of Prometheus allows for counting the per-second average rate of a time series' increase over a set amount of time. Here, the sum of each errors' rate is the average increase of errors per second over a rolling 5 minute interval. This is divided by the similarly rated increase of HTTP requests received. This gives the fraction of HTTP requests resulting in a network error.

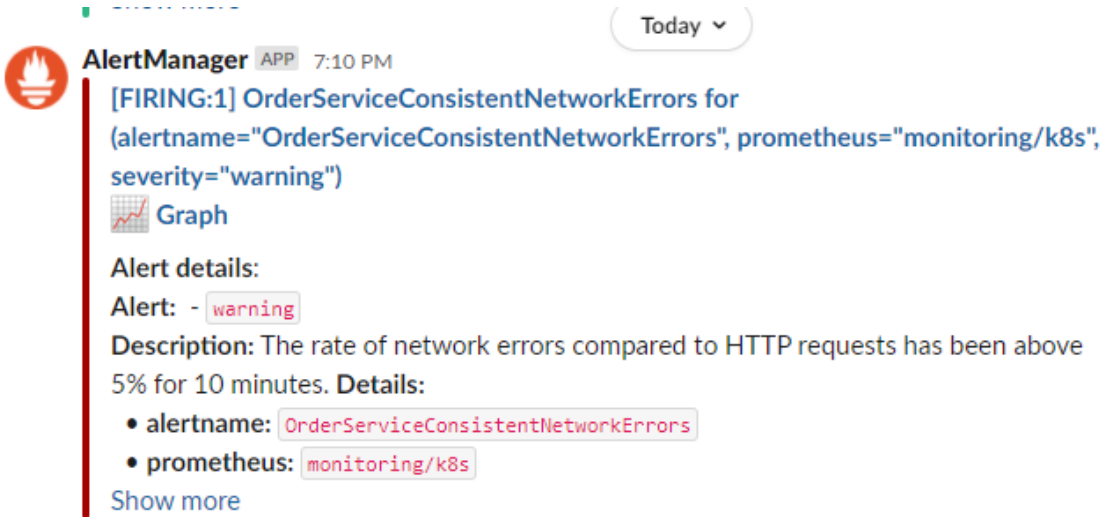

Figure 4.17: Alert firing message received inside Slack.

new one took its place next to the previous Pod. While this is happening, the operations and development teams responsible receive notification and start figuring out how the application can be better operated.

From annotation C) and on, a big spike in errors can be seen; however, this is due to another instance of the EventStoreDB launching and reconfigurations being made while still under load. This can be noticed in the Pod descriptions shown in Figure 4.20. The new Pod had to be restarted 3 additional times over the course of more than 8 minutes. This is due to the liveness probe function[13]. A similar thing can be noticed about one of the order service Pods to a lesser extent.

At annotation D), the system warms up and the error rate stabilises; however, it noticeably remains above the alert threshold. This means that the measures taken were not enough to lighten the load on the system. The decision is made that the already present database instances can be allocated more resources. At annotation E), the databases are given 200 millicores of processing power each, as shown in Figure 4.21, raising their individual CPU share to their double.

After a similar warm up period, at annotation F), the error rate finally drops below the 5% threshold and the alert sends a resolved status in Slack, as shown in Figure 4.22.

This does not mean that the problem is solved, nevertheless. The efforts taken to mitigate the errors were manual changes, and served only the purpose of keeping the application operable while the teams responsible were looking at permanent solutions. The operations team made the decision of allocating more resources to the database while keeping the replica number at its initial 1. As such, the operator got to work in the established CI/CDep system to push a new version of the cloud infrastructure to the version control system. This workflow is shown in Figure 4.23, the Deployment manifest is changed in one place: the CPU resource limit is raised to 200 millicores, double the original amount. At annotation G), this change runs through the automated CI/CDep pipeline and is deployed to the cloud platform in Figure 4.24.

[13]The liveness probe is a periodic request sent by the Kubernetes control process to a container to see if it responds. Under high load and when not using separate methods for these probes and other types of requests, a live Pod might be scheduled to be restarted. This is more due to the probe failing rather than the container being broken.

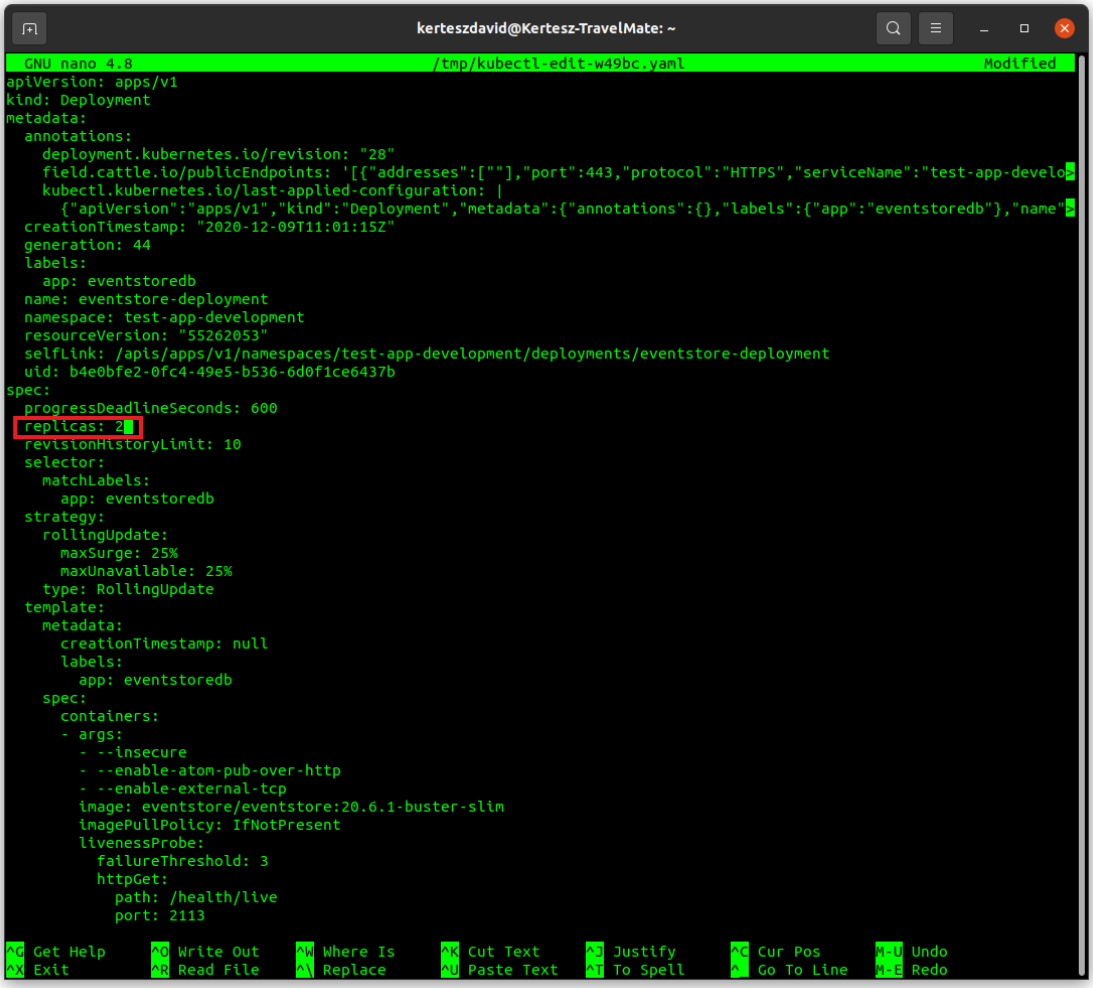

Figure 4.18: Scaling up of the EventStoreDB Deployment object by increasing its replicas with the changed values highlighted.

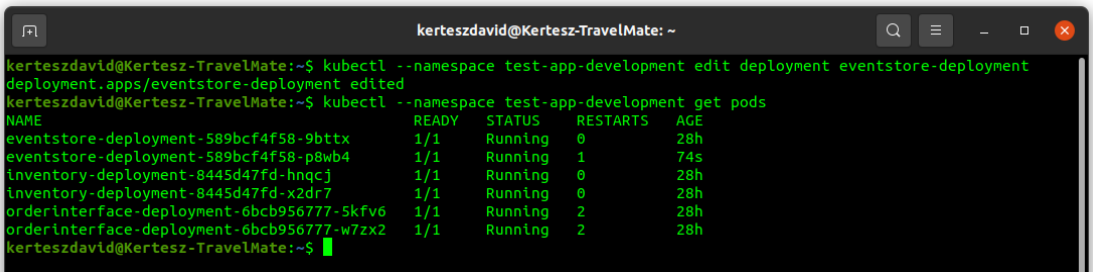

Figure 4.19: The "get pods" option used to see the status of the deployed Pods after the first kubectl edit.

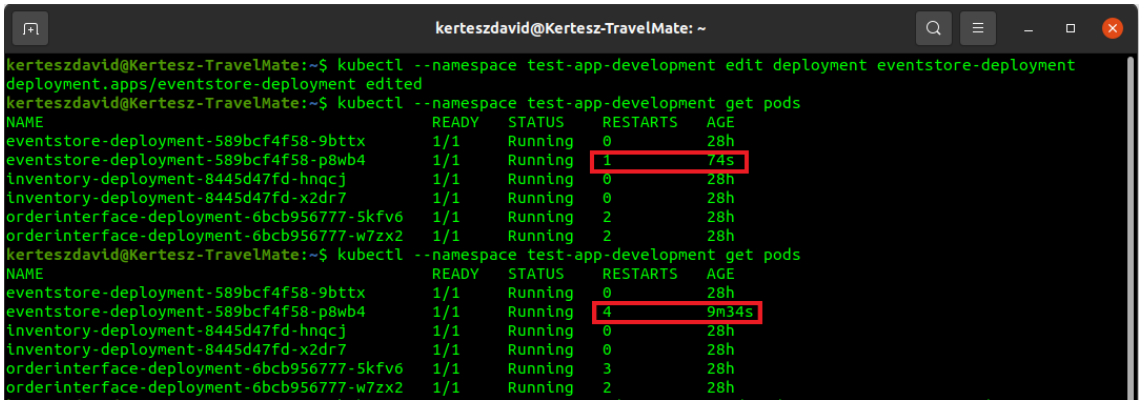

Figure 4.20: The "get pods" option used to check the functioning of the new Pods during the spike in errors after annotation C).

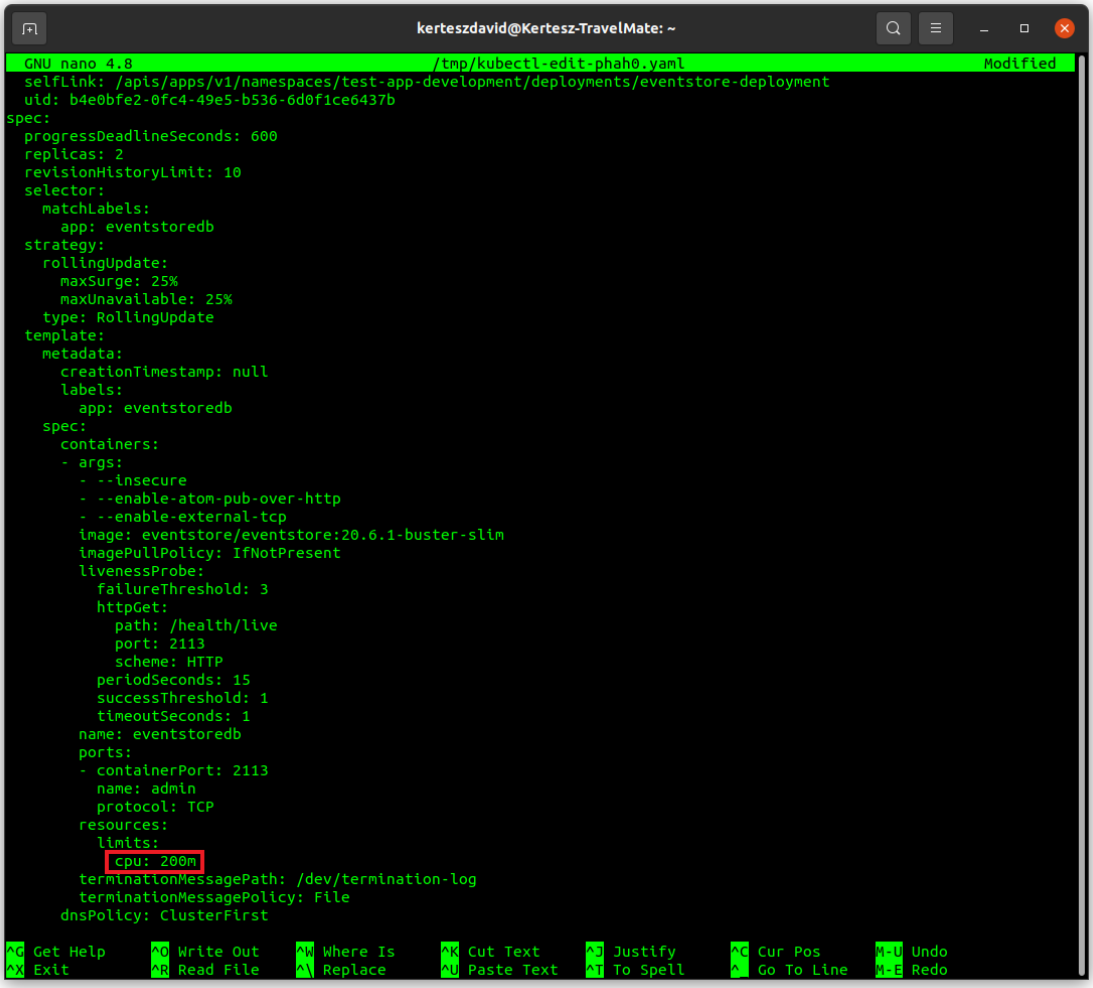

Figure 4.21: Scaling the processing power allocation of the EventStoreDB Deployment object with the changed values highlighted.

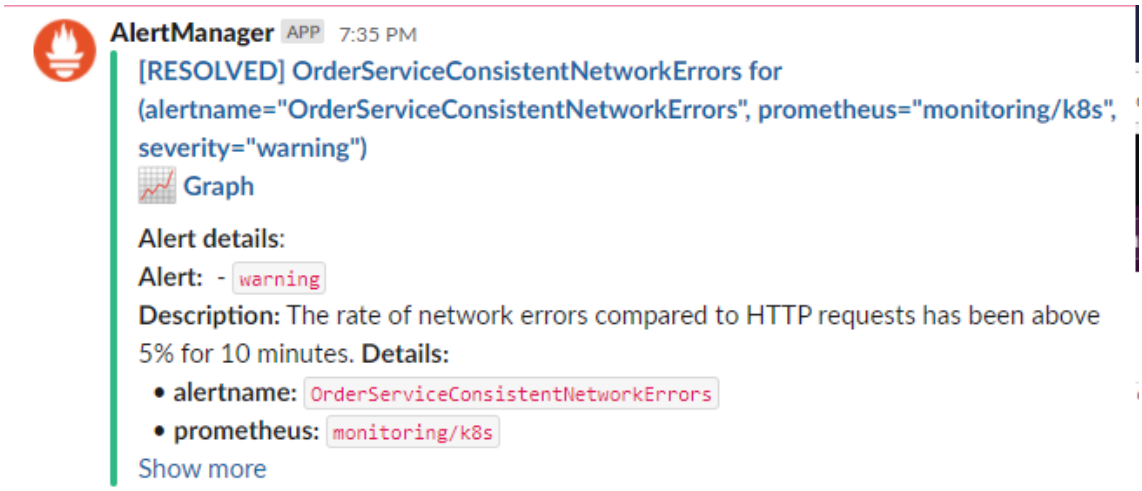

Figure 4.22: Alert resolved message received inside Slack.

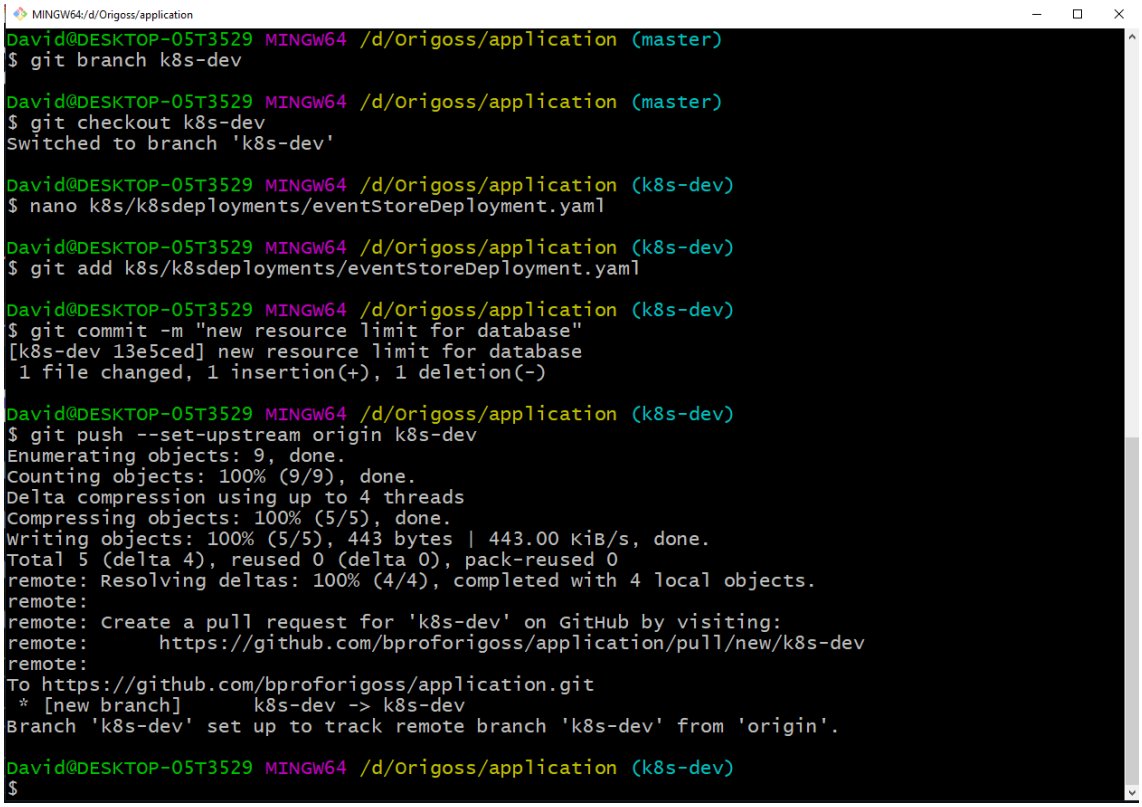

Figure 4.23: The git workflow of making changes to the Kubernetes infrastructure.

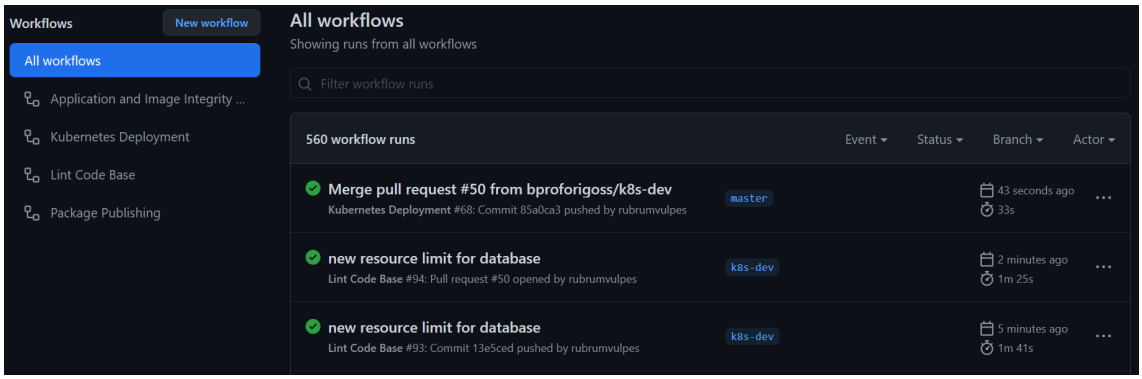

Figure 4.24: Dashboard of the GitHub Actions workflows showing the successful run of the merge from the k8s-dev branch.

By annotation H), the system warms up again and the error rate stabilises under the threshold again. While this was happening, the developer was looking through the implementation of the order service to find out if network errors could be caused by the service itself. What they found was that the timeout value of all connections related to serving incoming HTTP requests was set too low at 1. They made the decision to set it to 5 to allow more time for connections to form, even under load. This was carried out as shown in Figure 4.25. Similarly, at annotation I), the CI/CDep system performed checks and testing on the pushed changes and after success, it automatically published the new image version of the software and deployed it to the cloud platform as shown in Figure 4.26.

After the set amount of time, the hey tool returned with a report of the result of the requests it made to the path given, shown in Figure 4.27. The distribution of errors shows that the average error rate is representative of the success of the requests that were made. That is 522 291 successful requests to 233 269 unsuccessful ones, around 30% of all requests resulting in an error, factoring in the spikes and drops in error levels over the 60 minutes of the load testing.

In order to test the new system, a new round of load testing was begun with the exact same amount of load at annotation J). As shown in Figure 4.28, the changes made were effective, as the error rate stayed around 0 at all times until its conclusion at annotation K), except for a short spike that remained below the error threshold.

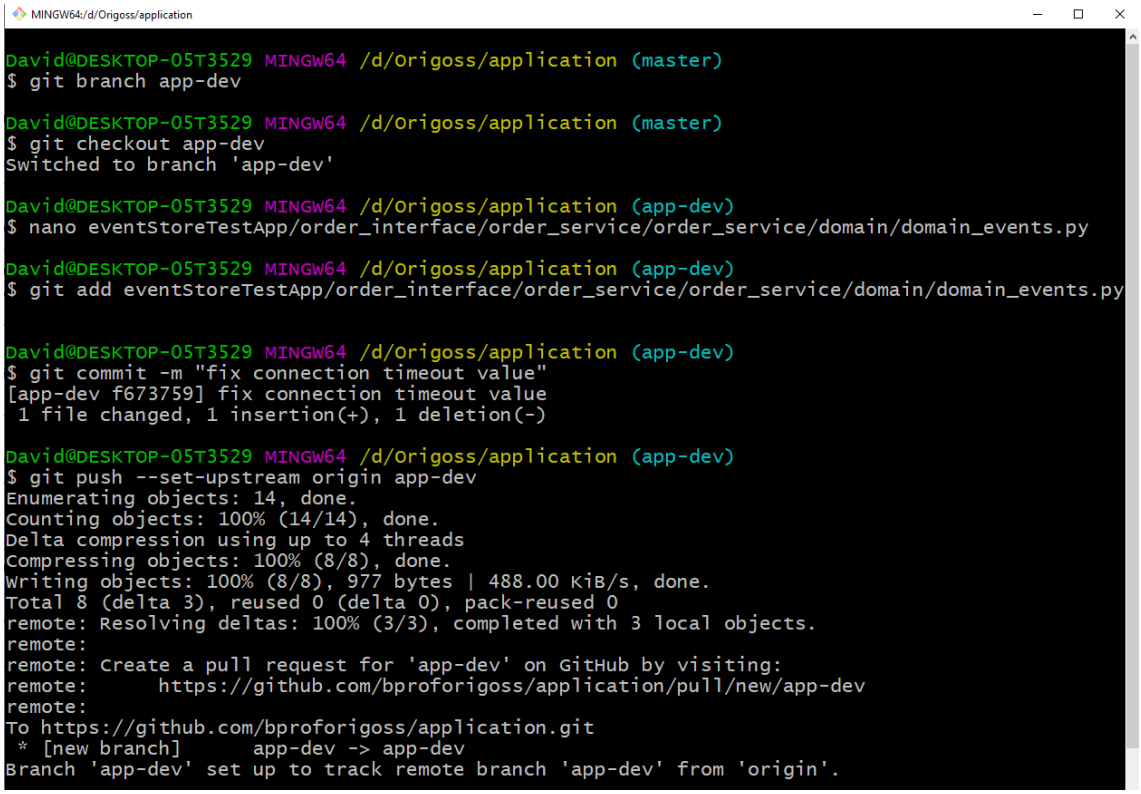

Figure 4.25: The git workflow of making changes to the order service implementation.

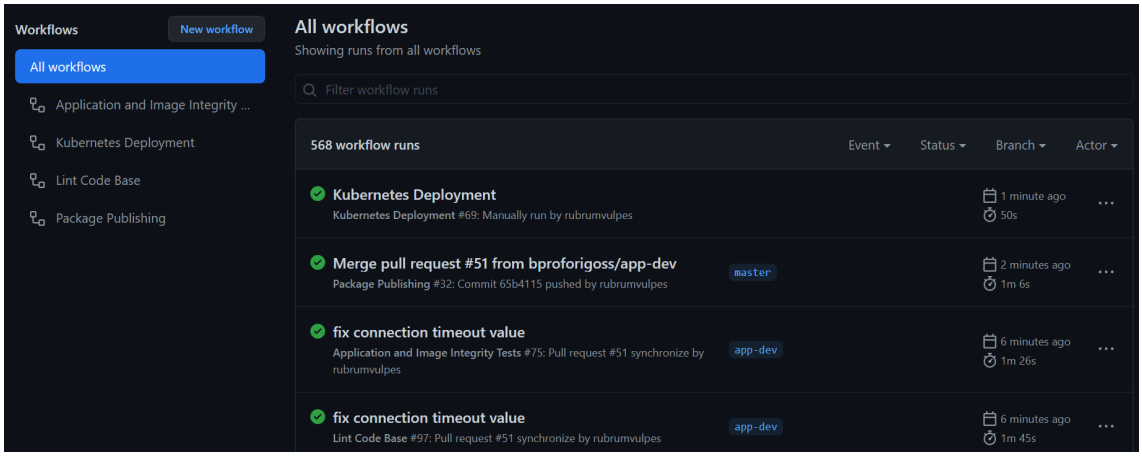

Figure 4.26: Dashboard of the GitHub Actions workflows showing the successful run of the merge from the app-dev branch.

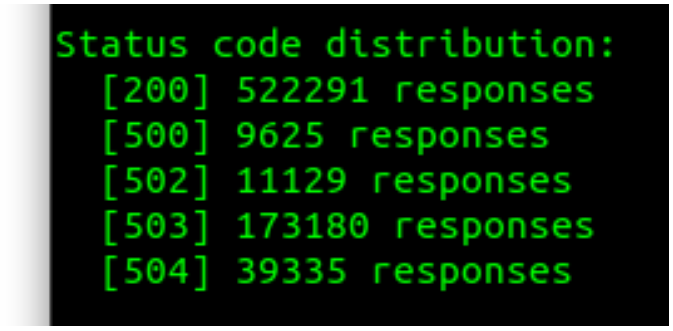

Figure 4.27: Error distribution provided by hey on the first test run.

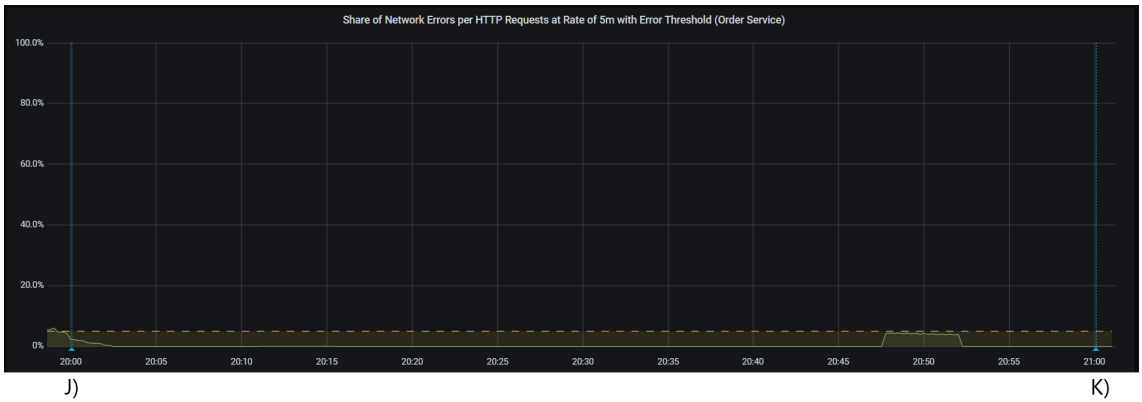

Figure 4.28: Error metric panel measuring the values used for alerting with annotations of specific events during the second load testing.

4.5.2 Observations

The scenario described in this section utilised many processes that have been implemented into the workflow of developing and operating the cloud-native platform and solution in this thesis. Two main observations have come up while running this test.

Firstly, the CI/CDep system and its ease of use. There are objectively well-performing aspects built into the DevOps platform that are used in this thesis, such as automated quality checks, unit and integrity tests, and automated deployment to the cloud platform. These tasks can take up a lot of time if performed manually. The CI/CDep system not only ensures that automated tasks perform without fault but also speeds up development. The load balancing test ran for an hour, this is including the start of the errors, the time to notice them, and the time to push the changes through the CI/CDep pipeline. Even so, the complete solution could be pushed in under that time, around an hour. This conforms to what the agile methodologies advocate, the quick incremental changes to software. Subjectively, the complex task of turning a whole cloud system around in case of a problem could not have been easier considering the many moving parts that have to be taken into account.

Secondly, the nature of cloud-native solutions. Kubernetes is one of the biggest technologies in the cloud-native ecosystem and its powerful capabilities enable operators to make significant changes to a cloud cluster with minimal effort. There were multiple times when a small change was introduced to the whole cluster during this test, and all was as easy as changing a line of code. More in-depth configurations would have to take other factors into account; however, the capabilities of Kubernetes enables its operators to quickly and easily enact changes in a complex cloud system with ease. The way changes are applied, only configuring resources when changes connected to it are noticed, also allows for extended automation. Except for developing the cloud infrastructure itself, there was no need for the developer to do anything related to Kubernetes or cloud administration. This makes it very easy to apply small changes without the need of extensive cooperation among personnel.

Chapter 5

Conclusion

The rapid evolution of cloud computing enabled developers and operators to provide highly available, on-demand services to customers worldwide. This evolution brought with it practices and methodologies that must be used in order to truly utilise the advantages of cloud computing. In this thesis, these methodologies, practices and cultural shifts were described.

DevOps was introduced by detailing its aspects, such as continuous practices or changes in the cooperation of teams from different areas. DevOps is not a collection of practices, it is a cultural shift spanning the whole width of the IT field. Its tenets stem from the cultural environment formalised in the Agile Manifesto. The authors of this manifesto envisioned a world where developers can have all the tools at their disposal to produce valuable and optimal software. With the growth of the online world, these demands came to the fore, and developers gained valuable practical experience in making web services. This learning curve culminated in many of the patterns described, such as the microservices architecture, the event sourcing pattern, or the Twelve-Factor Application. These interconnected aspects of the developing IT field makes up a very special ecosystem today that fully embraced the cloud evolution.

Kubernetes was introduced as one of the most dynamic and powerful orchestration solutions not only able to dynamically oversee a cluster of containerised applications but to be used in other fields of IT as well. From telecommunication to artificial intelligence, Kubernetes is a template that can be used to spur on research and development utilising the advancements in cloud computing. Since its open-source release by Google in 2015, Kubernetes has taken over the IT field and is now a mainstay with developers and operators alike.

This all culminates in what it is to be cloud-native. Subjectively, judging from the research that went into this thesis and the practical aspects that were measured during the load testing scenario, the cloud-native ecosystem holds simplicity and dynamism in the highest regard. The manner in which the utilised solutions interconnect and seamlessly operate as a single system is astonishing. To see so many software made for different purposes work so well together embodies what the open-source community, and through that the cloud-native ecosystem, is working towards: a world in which everyone has access to good software from talented developers to achieve anything they wish.

Bibliography

[1] History: The agile manifesto, 2001. URL http://agilemanifesto.org/history.html.

[2] What is DevOps?, 2021. URL https://aws.amazon.com/devops/what-is-devops/.

[3] 12 principles behind the agile manifesto, 2021. URL https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/.

[4] Alertmanager documentation, 2021. URL https://prometheus.io/docs/alerting/latest/alertmanager/.

[5] Thesis application repository, 2021. URL https://github.com/bproforigoss/application.

[6] CNCF cloud native definition v1.0, 2021. URL https://github.com/cncf/toc/blob/main/DEFINITION.md.

[7] Managing resources for containers - Meaning of CPU, 2021. URL https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

[8] [Illustration of the DevOps infinite cycle], 2021. URL https://dev.to/ashokisaac/devops-in-3-sentences-17c4.

[9] Periodic table of DevOps tools, 2021. URL https://digital.ai/periodic-table-of-devops-tools.

[10] What is FinOps?, 2021. URL https://www.finops.org/what-is-finops/.

[11] Flask documentation, 2021. URL https://flask.palletsprojects.com/en/1.1.x/.

[12] Gartner glossary - DevOps, 2021. URL https://www.gartner.com/en/information-technology/glossary/devops.

[13] [What is GitOps - git diagram], 2021. URL https://www.weave.works/blog/what-is-gitops-really.

[14] [Illustration of the pull-based deployment approach], 2021. URL https://www.gitops.tech/images/pull.png.

[15] [Illustration of the push-based deployment approach], 2021. URL https://www.gitops.tech/images/push.png.

[16] GitOps, 2021. URL https://www.gitops.tech/.

[17] Grafana documentation, 2021. URL https://grafana.com/grafana/.

[18] Loki documentation, 2021. URL https://grafana.com/oss/loki/.

[19] Loki description, 2021. URL https://grafana.com/docs/loki/latest/.

[20] What is DevOps?, 2021. URL https://azure.microsoft.com/en-us/overview/what-is-devops/.

[21] Nginx blog, 2021. URL https://www.nginx.com/blog/.

[22] Prometheus Python client GitHub repository, 2021. URL https://github.com/prometheus/client_python.

[23] Querying Prometheus, 2021. URL https://prometheus.io/docs/prometheus/latest/querying/basics/.

[24] Data model - Prometheus, 2021. URL https://prometheus.io/docs/concepts/data_model/.

[25] Prometheus documentation, 2021. URL https://prometheus.io/.

[26] [Prometheus Operator architecture], 2021. URL https://github.com/prometheus-operator/prometheus-operator/blob/master/Documentation/user-guides/images/architecture.png.

[27] Prometheus Operator documentation, 2021. URL https://prometheus-operator.dev/.

[28] Promtail documentation, 2021. URL https://grafana.com/docs/loki/latest/clients/promtail/.

[29] Pytest documentation, 2021. URL https://docs.pytest.org/en/6.2.x/.

[30] Understanding DevOps, 2021. URL https://www.redhat.com/en/topics/devops.

[31] What is DevSecOps?, 2021. URL https://www.redhat.com/en/topics/devops/what-is-devsecops.

[32] Requests documentation, 2021. URL https://docs.python-requests.org/en/master/.

[33] What you need to know - guide to GitOps, 2021. URL https://www.weave.works/technologies/gitops/.

[34] Event Store source code repository, 2021. URL https://github.com/EventStore/EventStore.

[35] Event Store website, 2021. URL https://www.eventstore.com/.

[36] hey documentation, 2021. URL https://github.com/rakyll/hey.

[37] Kubernetes user case studies, 2021. URL https://kubernetes.io/case-studies/.

[38] Kubernetes container runtimes documentation, 2021. URL https://kubernetes.io/docs/setup/production-environment/container-runtimes/.

[39] Kubernetes Deployment, 2021. URL https://kubernetes.io/docs/concepts/workloads/controllers/deployment.

[40] Kubernetes documentation, 2021. URL https://kubernetes.io/docs/home/.

[41] Understanding Kubernetes objects, 2021. URL https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/.

[42] kubectl documentation, 2021. URL https://kubernetes.io/docs/reference/kubectl/overview/.

[43] Kustomize documentation, 2021. URL https://kustomize.io/.

[44] K. Beck. *Extreme programming explained: Embrace change*. Addison Wesley, 1999. ISBN 9780201616415.

[45] K. Beck and C. Andres. *Extreme programming explained: Embrace change*. Addison-Wesley Educational, 2 edition, 2004. ISBN 9780321278654.

[46] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001. URL http://agilemanifesto.org/.

[47] E. Brewer. CAP twelve years later: How the rules have changed, 2012. URL https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/.

[48] J. Castro, D. Cooley, K. Cosgrove, J. Garrison, N. Kantrowitz, B. Killen, R. Lejano, D. Papandrea, J. Sica, and D. Srinivas. Don't panic: Kubernetes and Docker, 2020. URL https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/.

[49] DevOps Research and Assessment. DevOps capabilities, 2021. URL https://www.devops-research.com/research.html.

[50] E. Dietrich. There is no such thing as waterfall, 2021. URL https://daedtech.com/there-is-no-such-thing-as-waterfall/.

[51] V. Driessen. A successful Git branching model, 2010. URL https://nvie.com/posts/a-successful-git-branching-model/.

[52] E. Evans. What is DDD. Presented at Domain-Driven Design Europe. 2019, Amsterdam, NL, 2019. URL https://www.youtube.com/watch?v=pMuiVlnGqjk&t=1s&ab_channel=Domain-DrivenDesignEurope.

[53] M. Fowler. The new methodology, 2005. URL https://martinfowler.com/articles/newMethodology.html.

[54] M. Fowler. Event sourcing, 2005. URL https://martinfowler.com/eaaDev/EventSourcing.html.

[55] M. Fowler. Continuous integration, 2006. URL https://www.martinfowler.com/articles/continuousIntegration.html.

[56] M. Fowler. Continuous delivery, 2013. URL https://martinfowler.com/bliki/ContinuousDelivery.html.

[57] M. Fowler. Martin Fowler's professional blog, 2021. URL https://martinfowler.com/.

[58] M. Fowler and J. Lewis. Microservices: A definition of this new architectural term, 2014. URL https://martinfowler.com/articles/microservices.html.

[59] M. Fowler and D. Rice. *Patterns of Enterprise Application Architecture*. A Martin Fowler signature book. Addison-Wesley, 2003. ISBN 9780321127426.

[60] K. Hoffman. *Beyond the Twelve-factor App: Exploring the DNA of Highly Scalable, Resilient Cloud Applications*. O'Reilly Media, 2016. ISBN 9781491944035.

[61] M. Hofmann, E. Schnabel, K. Stanley, and IBM Redbooks. *Microservices Best Practices for Java*. IBM Redbooks, 2017. ISBN 9780738442273.

[62] B. Horowitz. Microservices Reference Architecutre, part 5: Adapting the twelve-factor app for microservices, 2016. URL https://www.nginx.com/blog/microservices-reference-architecture-nginx-twelve-factor-app/.

[63] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN 9780321670229.

[64] JetBrains. The state of developer ecosystem 2020, 2020. URL https://www.jetbrains.com/lp/devecosystem-2020/.

[65] M. Marchesi. The new XP, 2005. URL https://www.academia.edu/2862372/The_new_XP.

[66] R. C. Martin. The Future of Programming. Presented at First X/UP Meetup Organised by Expert Talks Mobile (Minute: 49:32). 2016, London, GB, 2016. URL https://www.youtube.com/watch?v=ecIWPzGEbFc&t=2972s&ab_channel=PaulStringer%27sMobileTech.

[67] Stack Overflow. 2020 annual developer survey, 2020. URL https://insights.stackoverflow.com/survey/2020.

[68] S. Prince. The product managers' guide to continuous delivery and DevOps, 2016. URL https://www.mindtheproduct.com/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/.

[69] C. Richardson. Microservices architecture, 2021. URL https://microservices.io/.

[70] C. Richardson. Command query responsibility segregation, 2021. URL https://microservices.io/patterns/data/cqrs.html.

[71] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, page 328–338, Washington, DC, USA, 1987. IEEE Computer Society Press. ISBN 0897912160.

[72] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 03 2017. DOI: 10.1109/ACCESS.2017.2685629.

[73] Y. Sundman. Continuous delivery vs continuous deployment, 2013. URL https://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment.

[74] C. Weisert. There's no such thing as the waterfall approach! (and there never was), 2003. URL http://www.idinews.com/waterfall.html.

[75] A. Wiggins. The new Heroku (part 4 of 4): Erosion-resistance & explicit contracts, 2011. URL https://blog.heroku.com/the_new_heroku_4_erosion_resistance_explicit_contracts.

[76] A. Wiggins. The Twelve-Factor App, 2017. URL https://12factor.net/.

[77] G. Young. Event Sourcing. Presented at GOTO Aarhus. 2014, Aarhus, DK, 2014. URL https://www.youtube.com/watch?v=8JKjvY4etTY&ab_channel=GOTOConferences.

List of Figures

List of Tables

List of Source Codes