Elasticity Concept for a Microservice-based System

MASTER THESIS

Aron Metzig

Submitted on 25 February 2022

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Georg Schwarz
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 25 February 2022

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 25 February 2022

Acknowledgements

I would like to thank my mother for giving me the opportunity to pursue a career in science and I would like to thank Thomas and Noah to make this path enjoyable.

For always kind and supportive suggestions, a special thanks for Georg Schwarz, who made this thesis possible.

Abstract

Software Elasticity is the concept of adapting available resources to the current or expected workload. This concept fits modern and stateless microservice architectures, which are scalable by design. Their scalability is closely related to Software Resilience and places new demands on cloud architectures. The JValue Open Data Service (JValue ODS) is an open data platform with focus on Extract, Transform, Load (ETL) pipelines and aims to make the usage of open data easy, reliable and safe. For the success of the ODS, an Elastic and therefore Resilient hosting is mandatory. This thesis deploys the ODS to an on-premise Kubernetes cluster to improve the uptime guarantee, discusses different deployment strategies, elaborates horizontal microservice scaling techniques and operates the necessary infrastructure. This thesis presents Peffer's Design Research Process to build a concept for Elasticity in microservice-based architectures. The concept is demonstrated and evaluated in the context of the JValue ODS.

Contents

1 Introduction **1**

2 Problem identification **3**
2.1 Elasticity . . . 3
2.2 Resilience . . . 4
2.3 Cloud . . . 5
2.4 Microservices . . . 7
2.5 Open Data Service . . . 8

3 Objective definition **11**

4 Solution design **13**
4.1 Container Orchestrator . . . 13
4.2 Template engine . . . 14
4.3 GitOps . . . 15
4.4 Service Mesh . . . 17

5 Implementation **19**
5.1 Kubernetes . . . 19
5.2 Helm . . . 23
5.3 Kubernetes distributions . . . 24
5.3.1 Development cluster . . . 24
5.3.2 On premise cluster . . . 25
5.3.3 Hybrid cluster . . . 31
5.4 Cluster ecosystem . . . 32
5.4.1 Monitoring . . . 32
5.4.2 Network file system . . . 33
5.4.3 GitOps . . . 34
5.4.4 Service mesh . . . 38
5.4.5 TLS certificates . . . 40
5.4.6 Autoscaler configuration . . . 41
5.5 ODS architecture . . . 42

5.5.1 Integration and production stage 42
5.5.2 RabbitMQ Cluster 43
5.5.3 Database Cluster 43
5.5.4 Services 45
5.5.5 Bootstrapping ArgoCD 50
5.5.6 External access 50

6 Demonstration **53**
6.1 Stages 53
6.2 Rolling Releases 54
6.3 Elasticity concept 56
6.4 Resilience concept 58

7 Evaluation **59**
7.1 External Acessible cluster 59
7.2 Elasticity Concept 60

8 Conclusion **63**

Appendices **65**
A YAML for Service and Deployment 67
B HPA with N\2, U\60 68
C HPA with N\2, U\80 69
D HPA with N\5, U\60 70
E HPA with N\5, U\80 71
F Cloc with Helm 72
G Grafana 73
H Longhorn 74

References **75**

1 Introduction

In cloud computing, Elasticity describes the optimal adaptive usage of available resources. Therefore, modern orchestrators claim to provide the difficult task of resource providing and withdrawing on demand. Besides Elasticity, Resilience is essential for every cloud computing scenario. Furthermore, modern applications running in cloud scenarios follow a decentralized microservice paradigm by splitting business logic within smaller development teams and relying on more powerful middleware. Additionally, the agile mindset alongside the DevOps culture creates very dynamic and complex service meshes (Herbst et al., 2013).

The JValue ODS is such a modern and agile application. As a competitor for the leading Open Data Platform, scalable deployment in an elastic cloud environment is essential for overall success. In previous work, the ODS was hosted via a docker-compose configuration, a sufficient tool for single-server setups with no Elasticity concept and only minimal Resilience support. This thesis deploys the ODS into an on-premise Kubernetes cluster.
Kubernetes is an orchestrator that claims to handle distributed systems' complexity while combining them with industry-grade uptime guarantees. The established cloud computing platform is capable of hosting more than just the production instance, adds a staging environment, automatizes day-to-day administration tasks and provides mesh-tracing and monitoring tools that support the further development of the ODS.

Peffer's Design Science Research Methodology provides the framework for this engineering work. The DSRP is illustrated in figure 1.1 and the thesis is therefore structured as follows:
Starting with the problem identification, followed by the derived objectives and the design of the solutions. The implementation section follows a top-down approach: starting by allocating the server cluster followed by establishing a cluster ecosystem and deployment strategies for the individual ODS services. The Elasticity and Resilience concept will be tested using benchmarking and chaos engineering in the following demonstration section. The performance of the test results are discussed in the evaluation section. The conclusion discuss further improvements of the ODS Elasticity and further cluster extensions.

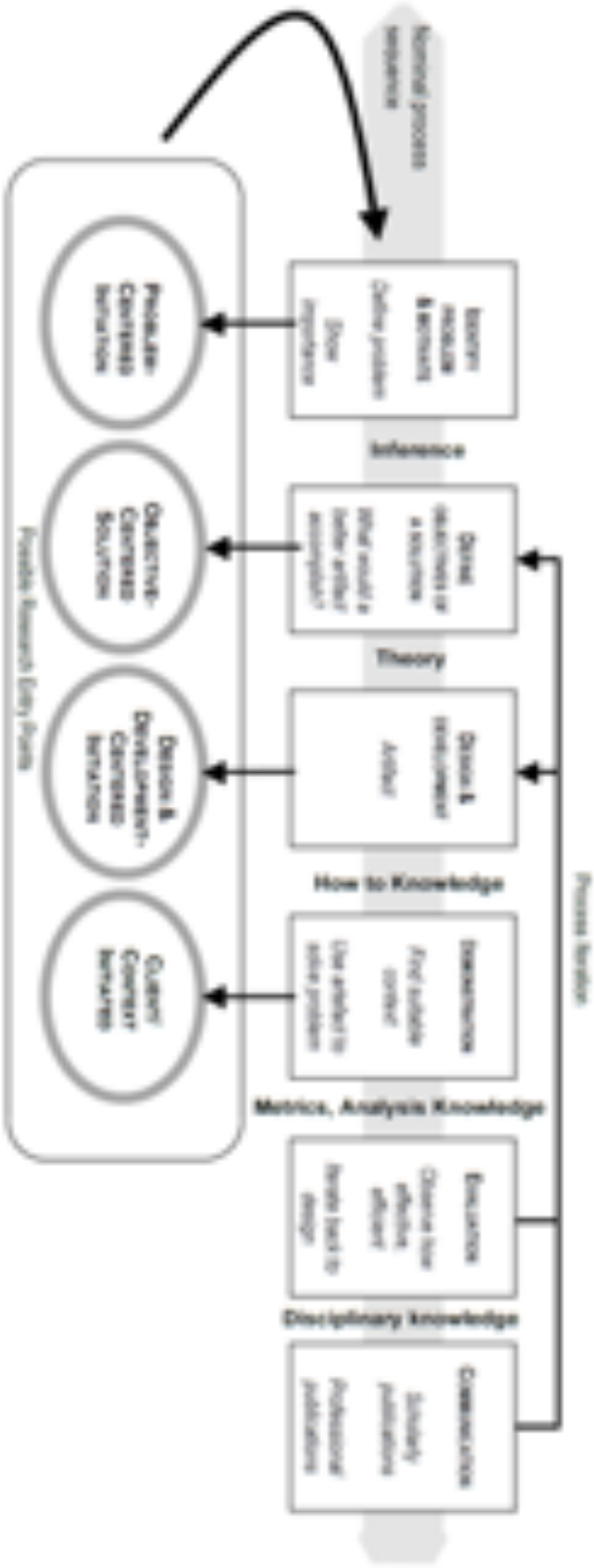


Figure 1.1: The DSRM, as used evaluation paradigm (Peffers et al., 2007)

2 Problem identification

The ODS is a production-ready application that requires integration into a modern cloud environment. Cloud providers like the Amazon Web Services (AWS), Microsoft Azure and the Google Cloud Platform (GCP) provide all kinds of toolchains and documentation for fast and easy deployment into their cloud. The easy deployment goes hand in hand with vendor lock-in, where the initial ease of application integration suddenly turns into unresolvable dependencies when systems must be migrated from one vendor to another. As commercial companies, cloud providers either can change their pricing strategy or an application can hit its free quota - resulting in higher operating costs as initially planned.

The ODS is an Open source software (OSS) that focuses on open data and is founded with public money - such a vendor lock-in is against its core values. Therefore, an on-premise solution is implemented by this thesis.

2.1 Elasticity

Herbst, Kounev and Reussner from Karlsruhe Institute of Technology published the paper 'Elasticity in cloud computing' in 2013. They transferred the term Elasticity from physics and economics into the field of cloud computing and also introduce the following definition that is adopted by this thesis:

Elasticity is the degree to which a system can adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time, the available resources match the current demand as closely as possible.

In order to be Elastic, scalability is necessary, which describes the ability of a system to scale up or down. The ODS follows the microservice design paradigm, which is explained in more detail in chapter 2.4. This stateless architecture is scalable per design and thus applicable to the ODS.

Validating Elasticity is mainly described with two values: *speed* and *precision*. Here *speed* is the average time needed for the system to scale to an optimal

state, and *precision* is the absolute deviation from the current to the optimal state. For real-life applications, overprovisioning implies higher operating costs and underprovisioning insufficient customer experience. Also, more than one business application can run in a given cluster. Therefore, as shown in Figure 2.1, overprovisioning a resource can lead to stealing resources for another service, ultimately leading to a state of cluster-wide underprovisioning.

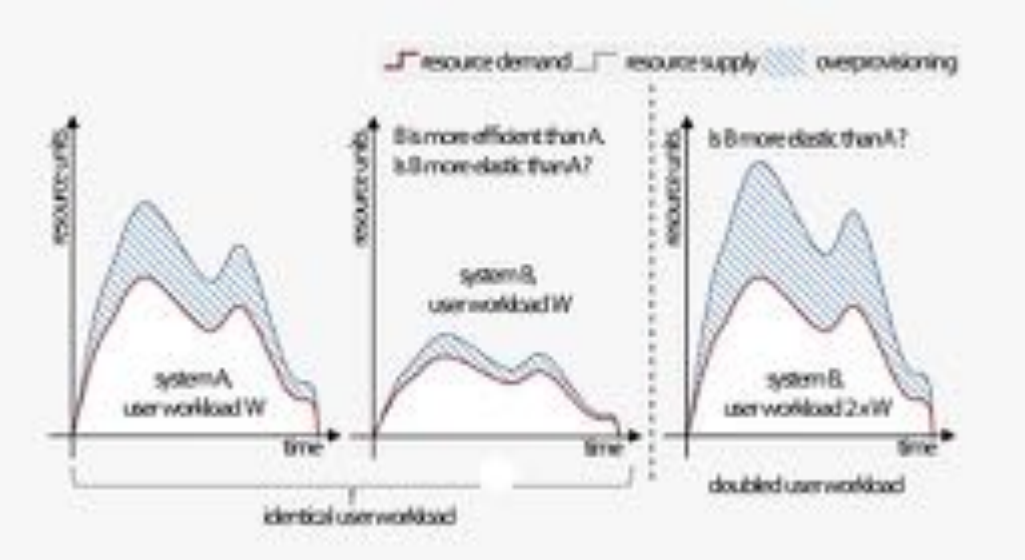


Figure 2.1: Elasticity and overprovisioning (Herbst et al., 2013)

2.2 Resilience

In the context of cloud computing, Resilience is closely related to Elasticity. Laprie's standard definition declates Resiliency as the persistence of service delivery that can be trusted justifiably when facing changes. Therefore, cloud Resiliency implies the extent to which a cloud system withstands the external workload variation and under which no computing resource reprovisioning is needed (*precision*) and the ability to reprovision a cloud system in a timely manner (*speed*) (Laprie, 2008; Ai et al., 2016). This matches the earlier definition of Elasticity in chapter 2.1.

This affects the scope of this thesis, as implementing an *Elasticity* concept implies creating a *Resilience* concept.

Cloud architects achieve Resilience with a combination of failbacks and redundancy, scaling between error handling of the implemented service and backups of whole datacenters. Resilience is even a cross-cutting concern, as layers may or may not need information about the robustness of the surrounding layers (Welsh & Benkhelifa, 2020).

Validating architectural Resilience is a non-trivial task as cross-cutting concerns cannot get isolated by their nature. Furthermore, it is nearly impossible to reproduce the same system state in complex, distributed, and elastic cloud scenarios. Since the scheduler could be running a task on another machine, a leader replica might have failed, the new cluster might not be in a strongly consistent state or the services themselves might be stateful.

This thesis uses traditional benchmarking alongside with so-called 'Chaos engineering' to validate the results. This term was brought to a broader audience by Netflix in 2011 and implemented with the Chaos Monkey, which is described as follows:

'The monkey randomly rips cables, destroys devices and returns everything that passes by the hand. The challenge for IT managers is to design the information system they are responsible for so that it can work despite these monkeys, which no one ever knows when they arrive and what they will destroy. Simulating failing applications, networks or servers.' (Evans, 2017)

2.3 Cloud

As early as 1962, John M. MacCharty already predicted that computing power could be sold like electricity and water. At that time, mainframes were the predominant structure of processing significant data streams. However, a mainframe was not designed to share its resources, it was built, maintained and operated by a single company.

Cloud providers closed that gap. They build and maintain infrastructure while selling them to customers. The following sales options are common:

- **Infrastructure** as a Service (IaaS) consists of rented (bare-metal) servers maintained by the tenant. The tenant has complete control here but needs to take care for all security-related issues such as firewall and software updates.
- **Platform** as a Service models (PaaS), the tenant has no access to the server's shell; Instead, he rents a configured instance of a deployment platform he is responsible for. Such a platform is usually capable of user management, access rights and application hosting.
- **Software** as a Service (SaaS) is the weakest of the three models, because only of an application hosting is provided. But this also includes preconfigured security features and snapshots of stateful services.

These terms get softened in the product mix of cloud providers. As an example, AWS provides so-called Spot-Instances, where one can rent unscheduled comput-

ing time for a discount. After 15 minutes of runtime, the machine can suddenly get switched off when needed somewhere else. Technically this is renting infrastructure, but for industry use-cases, those instances are mainly used for CI/CD pipelines, which are closer to a PaaS model.

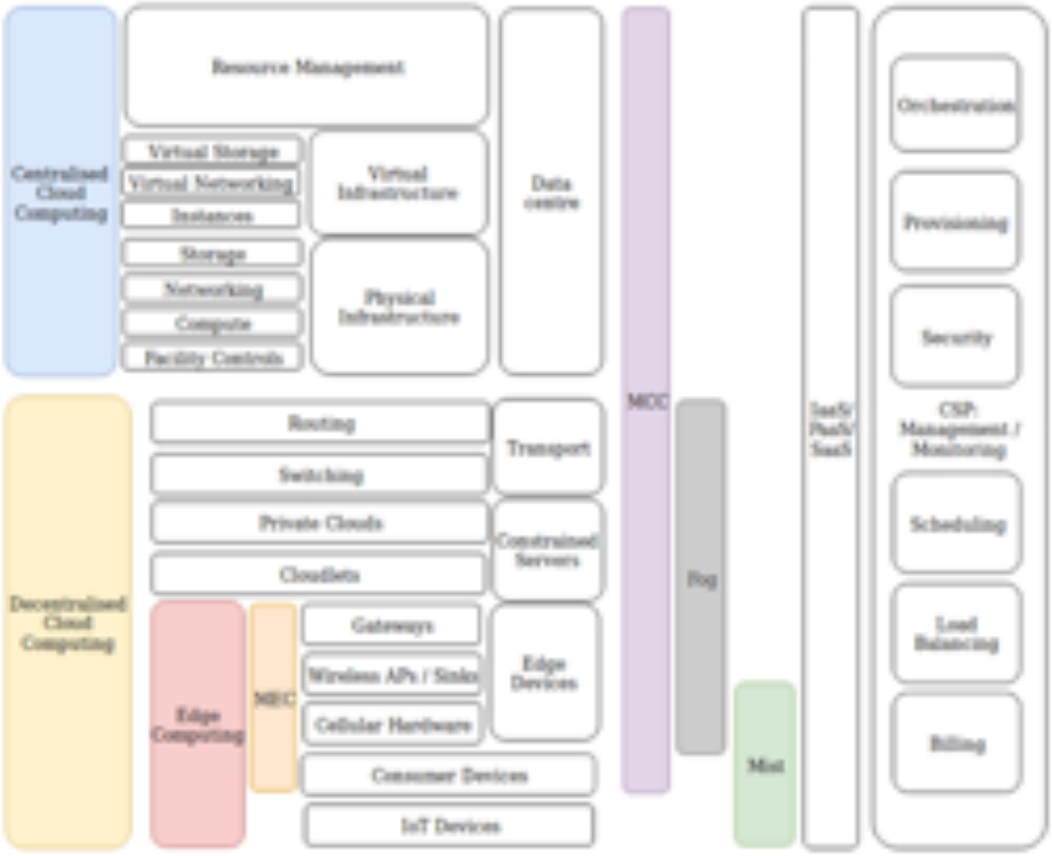


Figure 2.2: Modern cloud architecture (Welsh & Benkhelifa, 2020)

When discussing cloud integration, it has to be distinguished between edge and fog scenarios. In an edge scenario, the cloud services are the user's direct endpoints (living on the edge). A fog adds an additional layer for cumulating, normalizing and cleaning the data before storing them in the cloud (Welsh & Benkhelifa, 2020). In figure 2.2 the architechture one of these modern clouds are shown/demonstrated.

Clouds are designed to be Elastic. Since already elaborated, a system that is Elastic needs to be scalable and clouds implement two major scaling strategies:

- **Horizontal autoscaling** replicates the service, which can no longer guarantee that it can cope with its workload due to insufficient system resources. Services are stateless by design, hence replicating this service and splitting

the available resources, the workload on each available service gets balanced. As a result, the system is capable of using its resources more efficiently and can stay more responsive.

- **Vertical autoscaling** scales the available system resources by allocating new resources, like an external server, into the existing cluster. Adding a server from another cloud into a hybrid cloud environment is called a cluster outbreak.

This thesis initially started with 4 VMs with a bare Debian 9; additional 7 VMs were provided during the work - there are no further resources to allocate. The problem derived from this condition is the following: Dynamically allocate server (IaaS), configure the Elastic concept to horizontal autoscaling (PaaS) and host the ODS (SaaS) in an edge hosting scenario.

2.4 Microservices

The traditional software paradigm of monolithic software is changing. With GitOps and the agile mindset, faster delivery cycles, stricter role isolation and the need for scalable architectures in the context of cloud computing, microservices have established themselves. These loosely coupled and distributed systems allow development teams to *focus* more on their code base without losing *flexibility*. The service mesh of figure 2.3 underlines the importance of those two values. Because a single person can no longer understand the entire system architecture. On the other hand, are traditional monolithic systems too coupled for quick implementation changes - besides any technical boundaries, like compile times, dependencies and deployments (Nadareishvili et al., 2016).

In terms of Elasticity, microservices provide way better scalability than monolithic architectures, as horizontal scaling replicates the application microservices and has a lower memory and CPU footprint. Thus this design approach allows to replicate the services even more and make them more scalable. Additional microservice tend to have lower startup and tear-down times, which has a direct impact on the Elasticity's *speed* (Hilbrich, 2019).

Those benefits do not come for free. The communication between the services is not via traditional Interprocess communication (IPC) via shared memory models. As native members within the cloud, they inductively rely on their underlying technologies, like REST and GraphQL or gRPC. Shared workloads, events and even business logic are shifted into Message Orientated Middleware (MOM), like Apache Kafka and RabbitMQ.
Such MOMs build the core of distributed middleware, but are also an additional Single Point of Failure (SPOF). For this reason, MOMs are designed to operate in High Availability (HA) clusters, which makes them Resilient.

Figure 2.3: Netflix Architecture by Evans (2017)

Due to the microservice-based design of the ODS, there is no further need to elaborate an Elasticity concept; instead, every MOM, database or stateful service must be set up in a resilient way to avoid losing the Elasticity.

2.5 Open Data Service

The JValue Open Data Service is currently in its third major iteration. The first implementation was a monolithic architecture. In the second iteration, the monolith got split up into Spring Boot microservices for the third iteration. In the third and current iteration, most of the Spring microservices were abandoned and rewritten as node.js projects.

PostgreSQL is the chosen database management system. Some services rely on native features of this database, especially the Write-Ahead Log (WAL). The same applies for RabbitMQ. The services use native features like queuing, besides the common publish/subscribe pattern.

ODS currently has six microservices:

- **Adapter Service**

  The first step of an ETL, the Extraction, is performed by this service. It is the only Spring boot application responsible for fetching data from various data sources. Therefore, the service can process a wide variety of protocols and includes robust schema validation. The adapter service is also responsible for storing the data and provides a create, read, update and delete (CRUD) REST interface.

In order to stay stateless, this service uses the Outboxer pattern. For that reason, it is linked to an additional component, which uses PostgreSQL-specific streaming features, allowing atomic and unique processing of incoming messages.
As an extra service, its configuration is handled by environment variables, making it wholly configurable and portable for other microservices.

- **Notification Service**

  The notification service sends user notifications after triggered pipelines, by triggering webhooks, Firebase mobile notifications or Slack messages.

- **Pipeline Service**

  The second and most complex ETL step, the transformation, is handled in this service.

  It is implemented by submitting a plain Javascript function-string, alongside to the to data to be transformed. The function has a runtime limit of five seconds before it gets terminated. The pipeline service provides a REST interface to access its functions.

  This service also makes use of the previously explained outboxer pattern.

- **Scheduler Service**

  This singleton service schedules each step of the ETL process. The triggers are accessed via the services REST API, and therefore the scheduler has the most knowledge about the other services.

- **Storage Service**

  The Storage Service is accountable for querying and processing the stored data, by wrapping the database behind a PostgREST interface.

  Besides that, the Storage-MQ service is accountable for holding configuration options for pipelines. As being an event-driven service, the Storage-MQ service also goes along with the outboxer pattern.

  This service also includes a Liquibase schema version control for the database.

- **UI Service**

  The UI Service is an NGINX hosted Vue.js frontend for accessing the REST APIs of the cumulated ODS. The interface design language is inspired by Googles Material Design, and all of its components follow this pattern.

Hosting the ODS services is not a generic task. Some services rely on the outboxer or other middleware the scheduler service needs to be a singleton, while all others can be replicated and the PostgreSQL Database and the RabbitMQ must run in HA clusters.

3 Objective definition

According to the DSRP, the research goal and derived objectives were already defined at the first meeting to avoid confirmation bias. The objective are rewritten as Epics and assigned User Stories. Each User Story has a Definition of Done (DoD), which is used for evaluation in chapter 7.

- **EC-1**: As a User, I want to be able to access the JValue ODS with a URL
  - **EC-1.1**: As Developer, I want a comparison of available orchestration solutions

    DoD: List of requirements for orchestration solution, comparison by 'fully fulfills', 'partially fulfills' or 'does not fulfill'
  - **EC-1.2**: [Technical] As Developer, I want to develop the ODS in different repositories

    DoD: At least two repositories as image sources are handled
  - **EC-1.3**: [Technical] As Developer, I want to seperate the orchestrator infrastructure code from application code

    DoD: External repository for deployment files
  - **EC-1.4** As a Developer, I want to use a template engine to deploy the Infrastructure as Code (IaC)

    DoD: A template Engine generates generic template code
  - **EC-1.5** As a Developer, I want to deploy the services in a test stage

    DoD: Access to independent application stages inside the cluster
  - **EC-1.6** As Product Owner, I want to deploy with zero downtime

    DoD: Analysis of the deployment strategy
  - **EC-1.7** As Developer, I want to monitor the cluster application

    DoD: Visualize the deployed services and replicas

  - EC-1.8: As Administrator, I want to monitor the cluster

    DoD: Accessible Dashboard with essential cluster information

  - EC-1.9: As Administrator, I want to add a Server to the cluster

    DoD: The new server can process workloads

  - EC-1.10: As Administrator, I want to remove a Server form the cluster

    DoD: The cluster is in a healthy state without the removed server

  - EC-1.11: [Optional] As Administrator, I want to expand to a Hybrid cloud setup

    DoD: Prototype with determined time to active and costs

- EC-2: As User, I want the ODS to be able to stay accessible, even when processing heavy loads
  - EC-2.1: As Developer, I want to know how replication affects Elasticity

    DoD: Quantified analysis of replication approach

  - EC-2.2: As Developer, I want to know how sharding affects Elasticity

    DoD: Quantified analysis of sharding approach

  - EC-2.3: As Developer, I want to know how resource allocation affects Elasticity

    DoD: Quantified analysis of resource allocation approach

  - EC-2.4: As Product Owner, I want a general concept to scale up microservices

    DoD: The application can react to scale up with an increasing load

  - EC-2.5: As Product Owner, I want a general concept to scale down microservices

    DoD: The application can react to scale down with a decreasing load

  - EC-2.6: As Product Owner, I want a general concept for failsafe microservices

    DoD: The application can failover a simulated shutdown of services and server

- EC-3: [Optional] As Product Owner, I want Serverless user-code-Execution

4 Solution design

In order to fulfill all defined objectives, an extendable system needs to be implemented. Since all the defined objectives are relevant to every internet company, which are not using PaaS or SaaS models, the concept of container orchestration is state of the art.

4.1 Container Orchestrator

Running containers within Docker is an everyday task in the life of a software developer. However, containers were already known before the emergence of Docker in 2013. For instance, the Googles container-orchestrator Borg already used the concept of containers with the cgroup function of the Linux kernel before. With the success of Docker, the knowledge and possibilities of containers became accessible to everyone, and containerization became the standard for companies of all sizes (Verma et al., 2015).

To administrate and scale these containerized applications, container orchestrators got popular, for instance, Facebook's Tupperware, Docker Swarm, Google's Kubernetes and Apache Mesos. These orchestration tools attempt to abstract the underlying servers and hardware into a declarative description. These Infrastructure as Code based orchestrators usually underlie the promise theory, where the system guarantees you to retain the described state. A typical comparison for an orchestrator is with a post office:
One addresses a letter and drops it off at the post office, and the post office promises to deliver the letter within a day, no matter what happens in between ('Kubernetes Documentation', 2022).

Since containers and the resulting orchestrators have been in the IT stack of large enterprises for more than two decades, it can be deduced that they provide a competent infrastructure for Elasticity as well as Resilience.

4.2 Template engine

Due to the declarative approach of infrastructure, the same engineering principles apply to IaC and traditional application code. Hence infrastructure applications provide templates as applications provide design patterns. In application development frameworks are used to provide generic and recurring features; consequently, templates engines offer the same functionality for IaC.

To generate infrastructure code, templates engines have built-in support loops, if-else branches and dynamic variables. Furthermore, the resulting templates can call arbitrary container code, which themselves can execute further template generation. The generation of a parameterized and thus reusable base template is shown in figure 4.1 and adds A *don't repeat yourself* principle for IaC. In addtion, a set of templates is named a chart. A chart can hold its templates and further consist of sub-charts and libraries. Furthermore, sub-charts can access public variables and functions of the parent chart and libraries. The template's coupling result in adding the *Single-responsibility principle* and *Dependency inversion* paradigms.

```
--- # values.yaml
host: elastic-ods.tk
infraValues:
    - domain: argocd
      destination:
          host: argocd-server
--- # certs.yaml
{{ if .Values.infraEnabled }}
{{- range .Values.infra }}
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: "certificate-infra-{{ .domain }}"
spec:
  dnsNames:
  - "{{ .domain }}.{{ $.Values.host }}"
{{- end }} #end range
{{ end }} # End if
```

Figure 4.1: Helm chart with if, range and scoped variables

The principles described above are well-known practices for developing efficient and maintainable software. Consequently, applying clean code rules to templated charts is mandatory, as maintaining parameterized, generated code regarding of the programming language is a non-trivial task (Martin & Coplien, 2009).

4.3 GitOps

In the blogpost 'GitOps - Operations by Pull Request' of 2017, Weaveworks formed the term GitOps. In reference to DevOps, the well-known mixture of the words Development and Operations, GitOps is a mixture of Git and Operations. Due to its success, the CNCF adopted this paradigm with the OpenGitOps project. While still being in sandbox status, it is establishing a set of open-source standards, best practices, and community-focused education to help organizations to adopt a structured, standardized approach to implementing GitOps - proposing four core principles ('GitOps Principles v0.1.0', n.d.):

1. **Declarative**
   A system managed by GitOps must have its desired state expressed declaratively.
2. **Versioned and Immutable**
   The desired state is stored in a way that enforces immutability, versioning and retains a complete version history.
3. **Pulled Automatically**
   Software agents automatically pull the desired state declarations from the source.
4. **Continuously Reconciled**
   Software agents continuously observe the actual system state and attempt to apply the desired state.

As shown in section 5.1 and 5.2, Kubernetes altogether with Helm is already *declarative*, *versioned* and *continuously reconciled*. To achieve the last principle *pulled automatically* an external tool is needed. These GitOps tools are responsible for propagating the YAML defined state stored inside a Git repository into a Kubernetes cluster.

Architecture

Current software projects use multi-repo environments, which results in patches being distributed across multiple repositories to accomplish a single task. A new CI pipeline is triggered after each patch in every repository, so multiple pipelines with dependant, independent or even interdependent dependencies may run. Additionally, deployment pipelines are triggered by merges into specific branches, Git tagging or similar features not necessarily specific to Git. To not interfere with these already non-trivial and closely coupled pipelines, GitOps uses a pull- instead of a push-based system as illustrated in figure 4.2. Therefore, GitOps is an independent system and can also be utilized without any DevOps setup. In addition, any developer with push access to the repo can provide verifiable, declarative infrastructure patches and does not need direct access to the cluster or other credentials.

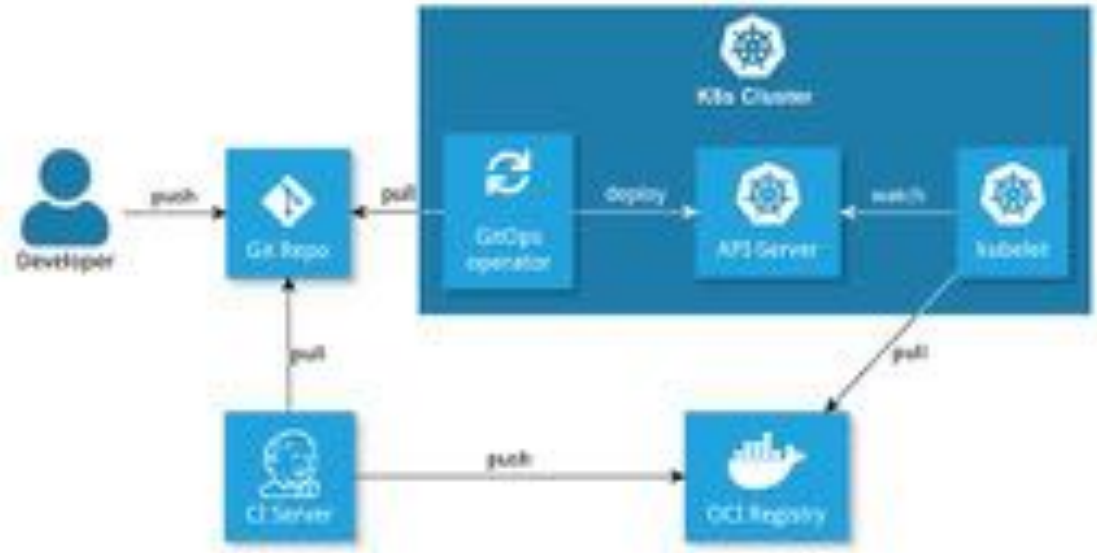


Figure 4.2: GitOps architecture (Johannes Schnatterer, 2021)

Disadvantages of this architecture are separated maintenance and versioning of application and infrastructure code, longer review spans across the multiple repositories and more complex local development. Those issues would get resolved in more advanced architectures, where the declarative infrastructure code is packed into the main repository and pushed together with the actual patch, the CI Server then pushes the infrastructure code to the previous, centralized Git repository ('Guide To GitOps', n.d.; Johannes Schnatterer, 2021).

This more DevOps-focused approach would remove an additional infrastructure repository for the developer and transfer the logic to the designated CI server. The drawbacks are Git conflicts caused by concurrency inside the CI pipeline, leading to inconsistencies. Some OSS libraries claim to solve this problem - unfortunately, the ODS uses Github Actions, where such a library does not exist yet. Thus the centralized mono-repo approach is the chosen implementation of this thesis.

4.4 Service Mesh

A service mesh is a dedicated infrastructure layer for controlling how different parts of an application share data with one another. This infrastructure layer can document the application's information flow, showing metrics for specific TCP/UDP or HTTP calls, and indicating failed service communication.

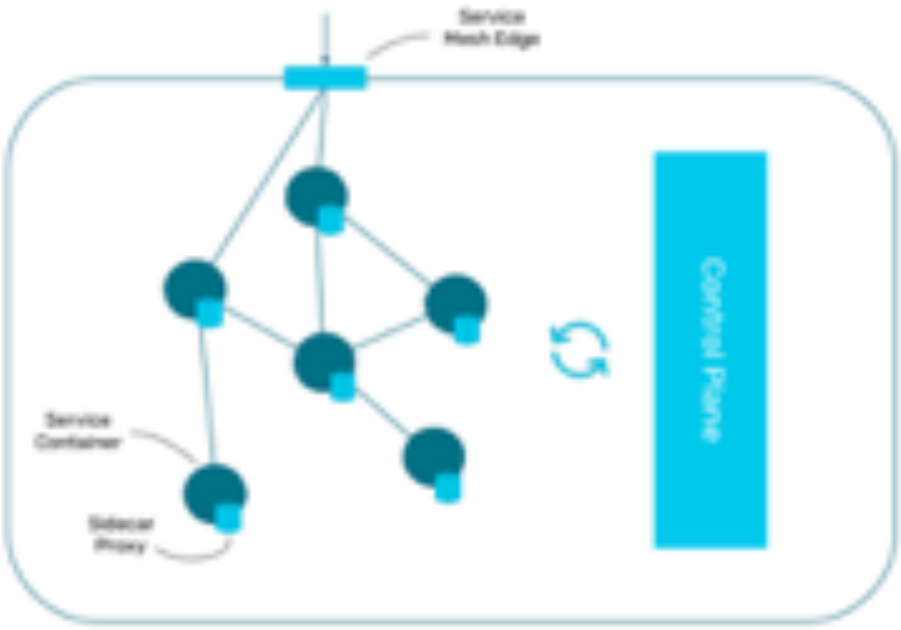


Figure 4.3: Kubernetes based mesh architecture (Kocot & Effing, 2021)

According to the survey 'Istio Service Mesh' in 2020, published by the CNCF, Istio is the leading service-mesh implementation, even though not being part of the CNCF. The CNCF has its own service graduated service mesh project: Linkerd. Both mesh work by applying a sidecar-proxy to every container. The term sidecar is used since they run alongside each service rather than within them. Decoupled from each service, the individual proxies form the observable mesh network.

Kubernetes has first citizen support for Istio as well as Linkerd, by injecting sidecar-proxies before the actual container startups. Once configured, the injection is handled fully automatically. Figure 4.3 shows how the service mesh is laid out inside a Kubernetes cluster.

Whether directly or indirectly, establishing a service mesh to the cluster and the ODS increases their Resilience. Components with direct impact are inter-Pod TLS communication, network policies, custom connection timeouts and even retry strategies for failed HTTP calls. More importantly is that the indirect

capabilities are created by benchmarking, debugging, and identifying possible bottlenecks, which make the cluster more Resilient and enable ODS developers to craft an overall better application.
It is usually tricky for developers to build an internal picture of the entire service landscape of distributed microservices, especially with MOMs like RabbitMQ. In most scenarios, a developer is tied to a specific service and has little knowledge of the other microservices. Showing a cumulative invocation graph helps the developer to understand the architecture, hopefully resulting in a more performant and maintainable system. All of this makes a service mesh an essential part of the Elastic concept.

5 Implementation

Starting with four VMs, an empty Git repository, and container image pipelines for the ODS - the steps to implement the Elasticity concept are: Select a container orchestrator, allocate a cluster, deploy the ecosystem and, finally, the ODS.

5.1 Kubernetes

'Kubernetes is a platform for building platforms. It's a better place to start; not the endgame.', was tweeted in 2017 by Kelsey Hightower, a Google Cloud advocate.

History

Google initially open-sourced Kubernetes in 2014. In 2015 with the release of Kubernetes 1.0, the CNCF was founded by the Linux Foundation alongside all noteworthy cloud providers worldwide. Since then the CNCF is responsible for Kubernetes and sponsors new technologies. Those projects usually are either a native component or a first-call citizen inside the Kubernetes ecosystem.

According to Github, the top 20 upstream maintainers are Google, Redhat, Microsoft, VMware and Goldman Sachs employees. *Commercial open source* projects like the Linux Kernel and MySQL are publicly developed and internally used by the involved companies. Such OSS is typically located in the lower parts of the IT Stack for decreasing the time to market and the overall product quality (Riehle, 2011).
'Google Kubernetes Engine' by Google, 'Openshift' by Redhat, and 'Azure Kubernetes Service' by Microsoft are products in the maintainer's portfolios and are rented, pre-configured Kubernetes instances. These instances appear to run recursively inside a Kubernetes platform. Kelsey Hightower's quote scales up.

The enormous success of Kubernetes, its diverse hosted applications and yet unmatched scalability make it the backbone of the cloud industry and thus the chosen orchestrator of this work.

Architectue

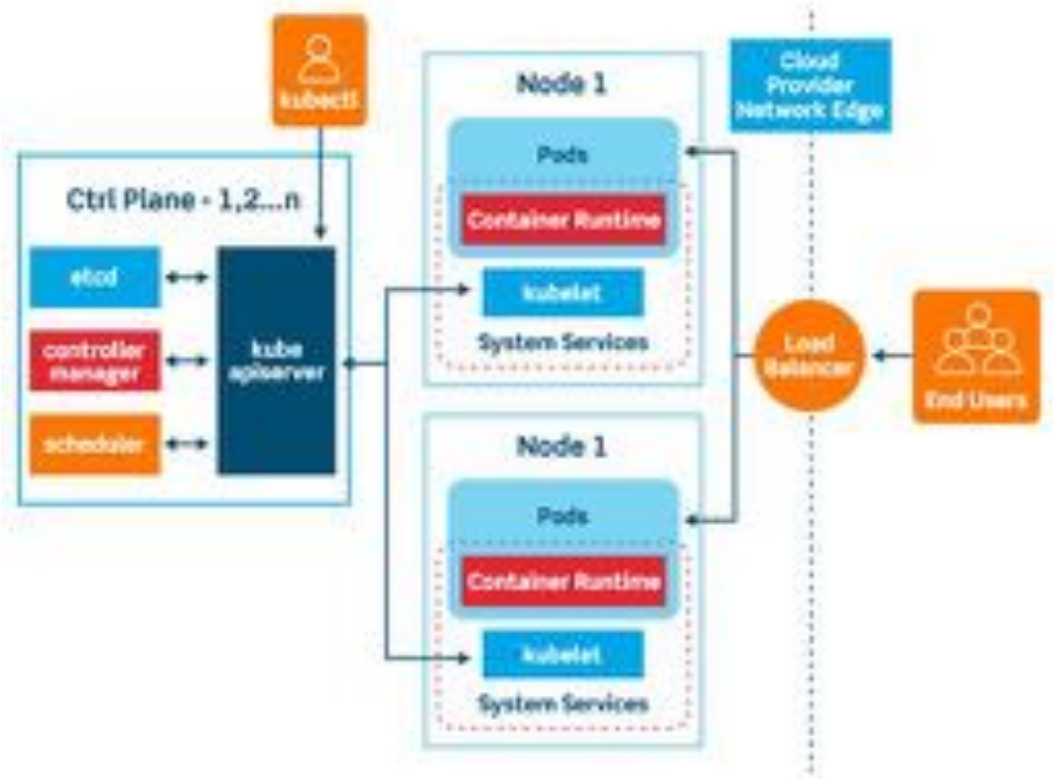


Figure 5.1: Kubernetes architecture (Chemitiganti, 2019)

Like every modern software system, the Kubernetes platform architecture is layered. These layers are all interchangeable, and some are still in active development and compete for the dominant implementation.

Most of the layers are shown in figure 5.1. The connection between the Nodes, Pods and the control planes are handled by the Container Network Interface (CNI), an interchangeable plugin.

The control plane resides in its own HA cluster, based on the primary Nodes. A Node in Kubernetes is either a physical or virtual machine. Primary nodes handle the internal cluster work, like hosting the control pane, scheduling, and load balancing. Agent nodes function as workers and are responsible for running the containers.

The control plane exposes the API and interfaces to define, deploy, and manage the lifecycle of containers. Its Single Point of Truth is an etcd database cluster. Etcd is a strongly consistent distributed key/value database, with CNCF certificated graduation state. For performance reasons, an etcd cluster typically

runs on the same machine as a Kubernetes primary node; this is called a stack topology. There is also an external etcd topology where the etcd cluster runs on external dedicated machines, which further increases Resilience.

It is important to note that the whole cluster fails when all control planes goes down. Running the primary nodes in HA mode, is crucial for an Elastic software concept.

Declarative Infrastructure

The behavior of Kubernetes can be defined entirely declarative by providing YAML files. To apply, reapply edited files or delete YAML-defined resources via the kubectl command is the most direct way a developer can interact with a cluster. The developer provides a YAML, which describes the desired state, and Kubernetes promises to match this state.

This thesis does not use any other vanilla Kubernetes components besides those explained above. Since Kubernetes quickly becomes complex, the explanation of the other components is skipped to avoid losing focus on the results and goals of the work.

Core components

Vanilla Kubernetes itself consists of many components. Each component handles its own abstraction layer, from manipulating and executing user code to representing the underlying hardware (Hightower et al., 2017).

A Pod is described as the smallest deployable unit deployed inside a cluster. One can imagine it as a currently executed docker image. Additional there are configuration options for networking, persisting, resource limits, and health checks. Each Pod requires a unique name, as it gets resolved via an internal Kubernetes DNS provider.

Kubernetes promises zero downtime features and self-healing with Deployments. In a Deployment, the desired state of some Pods are defined - for instance, a microservice with a replica count of two. When one of the Pods crashes, Kubernetes will immediately start a new Pod while the old one remains terminated. Deployments themselves are a facade pattern, as they do not handle the replication but instantiate so-called ReplicaSets, which function as the actual controller of the active Pods. Each Pod gets a unique name suffix to provide a unique DNS name for the Pods; thus, a Pod controlled by a ReplicaSet may be reachable with a name like ‘notification-service-5fc9b88494-nsd49‘.

To discover and communicate with the Pods’ names inside the ReplicaSet, Kubernetes introduces Services. They provide a single resolvable DNS name.

The Service DNS name would be just ‘notification-service‘. Besides the discovery aspect, Services also function as loadbalancers for their controlled Pods.

To avoid DNS name collisions over different development teams and projects, Kubernetes supports Namespaces. Namespaces are a way of subdividing cluster resources. The namespace domain is appended to each DNS name, e.g., 'notification-service.other-namespace', making the service easily accessible within the namespace without restricting access to services of other namespaces. Regular tasks, such as issuing TLS certificates, can be outsourced while remaining available to other services. In chapter 5.5.1 namespaces also will be used to deploy the ODS into a 'staging' and a 'production' environment.

Ingress

Ingress is a mixin of an L7 load balancer and a reverse proxy by applying custom rules. These rules can be extended - depending on the implementation of the Ingress controller - by regex path matching, URL rewrites, custom headers and other standard network tasks. To ensure performance, Ingress resides in the outermost load balancer of figure 5.1. From here, each request is routed through the controller pods, applying the defined rules and forwarding the (transformed) request to the internal network.

As the boundary between the internal and external network, forwarding HTTPS traffic and providing TLS certificates is also part of Ingress.

Horizontal Autoscaler

Kubernetes has an already integrated HorizontalPodAutoscaler (HPA). As this is a critical component for the Elasticity concept, and it is further explained in chapter 5.4.6.
An outline of its functionality is calculating the desired number of Pods and then applying the desired count by:

- starting a Pod and adding it to the service loadbalancer. The Pod is ready to process requests as soon as it is marked as healthy
- stopping a Pod, by no longer forwarding any requests to the terminating Pod, waiting for any dangling processed requests, and then releasing its used resources

Role Based Access Control and Operators

Kubernetes is an extendable platform accessible through a RESTful HTTP interface. For instance, retrieving a Pod in a particular namespace translates to the following HTTP call:

```
$ kubectl get pods -v=6
I0112 09:41:43.678008 846082 loader.go:372] Config loaded from file:
/home/knukro/.kube/config
I0112 09:41:43.697838 846082 round_trippers.go:454]
GET https://127.0.0.1:42969/api/v1/namespaces/default/pods?limit=500
200 OK in 12 milliseconds
```

The call can be mapped into a 'GET /API/GROUP/VERSION/RESOURCE' pattern. Pods, Nodes and any other core component are accessed via this resource-controlled interface, enabling a flexible REST-API with CustomResources. For instance, the CustomResource 'certificate' is, once created, accessible via the API by 'kubectl get certificate' and makes it a first-class Kubernetes citizen.
For security reasons, Kubernetes supports Role-based access control (RBAC). The Kubernetes Roles access control works by assigning rights for a particular resource in a specific group. For instance, granted access to read the Pod core group does not allow the same user to read the metrics group's Pod resource. Defining RBAC users and defining access rights is also done declaratively.

Operators are clients of the Kubernetes API which act as controllers for a Custom Resource. When deploying an operator, its custom resources are installed together with a container image for the operator code. When deploying the custom resource, the controlling operator creates the underlying VolumeClaims, StatefulSet and handles the initial configuration. On deleting the resource, the operator ensures that all dependent resources are deleted. Typically the software vendor provides a ready to use Kubernetes operator - ranging from RabbitMQ up to a GitLab instance.

5.2 Helm

Helm considers itself as the package manager for Kubernetes and is a graduated project of the CNCF.

Besides being a package manager, Helm is a templating engine for creating Kubernetes YAML ('Helm Homepage', n.d.; 'Helm Documentation', n.d.). The 'Go template language inspires this engine' supports loops, if-else branches, and dynamic variables for generating valid Kubernetes YAML files.

1. `helm install ods-certs -namespace default ./ods-certs-folder`
2. `helm list` shows `ods-certs` as `revision1`
3. Edit content in the *values.yaml*, e.g adding an element
4. `helm upgrade ods-certs ./ods-certs-folder`
5. `helm list` shows `ods-certs` as `revision2`

List 5.2: Helm workflow

Helm package manager features are not limited to adding and installing external libraries, but also managing the orchestrated YAMLs - including installing, listing and upgrading revisioned charts. A typical Helm workflow is shown in list 5.2, in step 3, the template is generated with the updated variables inside the *values.yaml*.

5.3 Kubernetes distributions

An unconfigured Linux is not a particularly useful Operating System (OS), depending on the use case one might choose to install initd, an XFCE desktop or disable the support for dynamically linked executables. Therefore, Linux distributions got popular to user groups with similar needs. A preconfigured distribution provides all tools to serve its generic or specialized purpose.
This principle translates to Kubernetes, whose original distribution is usually K8s. However, there are various distributions: Rancher, Minikube, Kind and k3s. Every distribution has a different target group and strengths and weaknesses. For consistency reasons, this thesis does not use the term K8s as a abbreviation for Kubernetes but always will use the name of the actual implementation.

A fundamental problem of Kubernetes clusters is bootstrapping. The bootstrap problem describes a self-starting process that is supposed to continue or grow without external input. For Kubernetes, this does not stop when being executed - TLS bootstrapping and GitOps bootstrapping are prominent examples.

5.3.1 Development cluster

It is essential to run and test the application code when developing, so it is no surprise that it is also an essential step for IaC. Thus a developer needs to bootstrap a local Kubernetes cluster. The following distributions are certified by the CNCF:

- **Minikube** is one of the oldest and most traditional implementations. Starting a single-node Kubernetes inside a small VM, which makes it independent but accessible for and from the host OS. Due to the restrictions of a needed hypervisor, starting and managing this type of cluster is comparably resource-intensive and slow.
- **K3s** is a smaller version of the original K8s implementations desgined for IoT and edge scenarios. It is considered lightweight and requires 50% less RAM compared to the original K8s implementation. The downside is a missing virtualization layer, as it is running as a system service. This can affect the developer's OS, especially when having little or no experience with cluster development.

- **Kind** is a single node cluster that runs inside of docker. With no other hypervisor between kernel- and userspace it is significantly faster than Minikube, but due to sandboxed resources, it is effortless to stop or reset a cluster after performing a prejudicial action.

In this work, *Kind v0.11.1* is used for local development tasks. Kind aims to be a minimal Kubernetes distribution, so there is no installed ingress-controller or preinstalled image registry. Since every development cluster likely needs ingress support, the convenience script *utils/kind_with_registry_and_ingress.sh* is provided and documented.

For local and productive monitoring and easy cluster access, *Lens IDE v5.3* is used. Lens is a powerful graphical interface for a fast overview of running pods and jobs, accessing logs or shell access, while also monitoring Elasticity relevant metrics and providing general cluster warnings.

5.3.2 On premise cluster

The official Kubernetes documentation is one of the best around, index- and fuzzy searchable, translated into nine languages and full of recommendations and examples.

When it comes to bootstrapping an on-premise cluster, it is surprisingly difficult to find the relevant chapters in the documentation. When stumbling over the installation steps with *kubeadmin*, this tutorial loses its elegance as fast as the provided convenience script *kops* installs a Kubernetes on an AWS machine.
Setting up a HA cluster, as described in chapter 5.1, is fundamental for this work and the official documentation is missing out on essential parts. For instance, it is not explained how to install and bootstrap important components: etcd cluster, DNS provider and loadbalancer.

Blogs, technical articles and even books usually assume an already running Kubernetes cluster and refer to either local single-node setups or renting a managed cluster of a chosen cloud provider. It becomes very apparent that K8s is a commercial open-source product, and cloud providers get their return of investment by selling their Kubernetes systems. For that reason, there is no interest in providing documentation for cluster bootstrapping but for typical development tasks.

Nonetheless, to set up an easy to administer on-premise Kubernetes, a k3s in HA mode is bundled together with a stacked, etcd cluster.

Etcd cluster

The evaluated stacked topology shown in figure 5.3 is based on an etcd cluster on each primary node. This will be initially setup on 4 VMs, divided into a primary and three worker nodes.

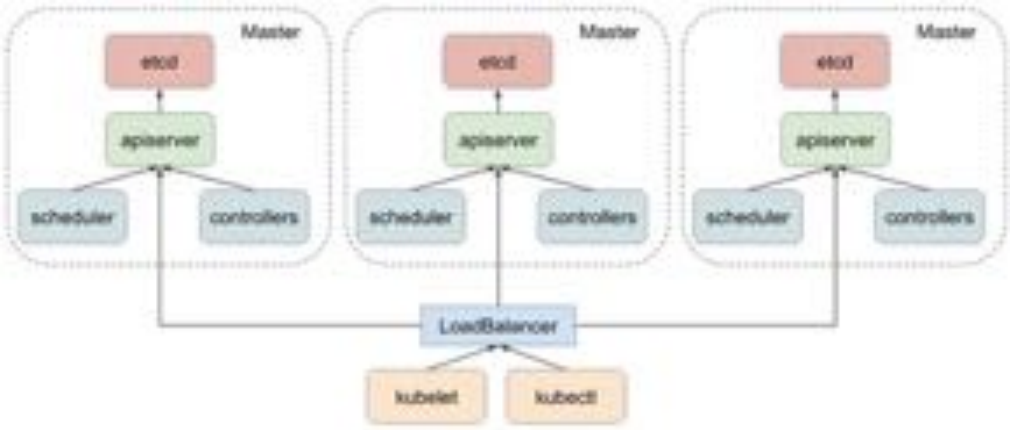


Figure 5.3: Stacked etcd topology (Bernaille, 2019)

There are other implementations for distributed key/value store, but as etcd is as mature as Kubernetes, it is the preferred implementation. Installing a single Etcd node requires the same steps like installing a multi-node cluster It is important to note that when performing an initial installation with N nodes, N must to be odd. Otherwise, the leader election will fail and the installation freezes.

The communication between and with the key/value nodes is done only via HTTPS. Because of this reason, a TLS certificate chain needs to be provided. We accomplish that by creating a self-signed root certificate. Each node then needs its member certificate and key, signed by the recently created root cert. The root-ca, the member certificate and key file are copied to the target machine. Enabling the inter-cluster communication is done via publishing its peer nodes in the following format: `etcd01=https://131.188.64.171:2380`. The IP configuration is published via the target machines public IP. Even for dedicated single node-setups, sticking to localhost would make the cluster not externally accessible.

Etcd can be run as a Docker image, but this would not make the installation portable and would introduce new netorkig and volume problems. Hence a traditional OS installation is chosen. To execute etcd, the binary must be downloaded and wrapped by a Linux system service. The created service provides the path to the executable and the deparametrized configuration is shown in figure 5.4.

Adding an etcd node to an existing cluster follows a slightly different pattern. The existing and healthy cluster first needs to be prepared via the tool `etcdctl`.

```
ETCD_NAME="etcd$INDEX"
ETCD_INITIAL_CLUSTER="$PEERS"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://$TARGET_IP:2380"
ETCD_LISTEN_PEER_URLS="https://$TARGET_IP:2380"
ETCD_LISTEN_CLIENT_URLS="https://$TARGET_IP:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://$TARGET_IP:2379"
ETCD_DATA_DIR="/var/lib/etcd"
ETCD_TRUSTED_CA_FILE="/etc/etcd/ssl/etcd-ca.crt"
ETCD_CERT_FILE="/etc/etcd/ssl/server.crt"
ETCD_KEY_FILE="/etc/etcd/ssl/server.key"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/ssl/etcd-ca.crt"
ETCD_PEER_CERT_FILE="/etc/etcd/ssl/server.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/ssl/server.key"
ETCD_PEER_CLIENT_CERT_AUTH="true"
```

Figure 5.4: Parameterized etcd cluster initialization configuration file

This tool must also be configured with the TLS toolchain to communicate with the cluster. The added machine only needs the IP of a single peer node and after starting the service, the cluster reconciles and is ready to use in HA mode.

K3s cluster

Like already mentioned, k3s is a certificated Kubernetes distribution. It is a single executable, designed for smaller IoT devices and Edge Computing, which matches the needs of the ODS. K3s installation claims to be straightforward via a convenience script that also supports high availability mode. Kubeadmin, on the other hand, needs significant knowledge above the target machine OS and thus manual configuration. Additionally, a Container Network Interface (CNI) (1) needs to be installed alongside a loadbalancer like metal3 (2).

1. CNI installation is done via a single, `kubectl apply` command, but depending on the implementation, further configuration is necessary.
2. Metal3 is an OSS L3 loadbalancer, certificated by the CNCF and needs declarative configuration, like the actual server IP - making it nontrivial to use.

K3s brings and configures its own CNI (Flannel) and loadbalancer (Klipper) as shown in figure 5.5, simplifying the setup and maintenance significantly.
Flannel is one of the oldest and fastest CNIs. Its downsides are missing security features, although this is compensated by using a service-mesh, enabling TLS encryption for inter-Pod communication (Ducastel, 2020). When running in a closed and fully controlled network, this attack vector is considered low impact.
Klipper on the other hand is a rudimentary loadbalancer implementation, but

the cluster size will likely not exceed a few hundred nodes, it is assumed to not affect the cluster performance. Furthermore, Klipper works out-of-the-box and requires no further configuration.

Figure 5.5: K3s architecture (K3s.io)

Installing k3s is straight forward, because they provide an install script, which is configurable via environment variables. As already stated, k3s can also be used as a single-node development platform and requires further configuration for HA mode. Due to the Kine module, shown in figure 5.5, k3s is capable of using different subsets of the etcdAPI for using SQLite, PostgreSQL, MySQL and dqlite besides etcd. Vanilla k3s uses either SQLite for single-node mode or dqlite in cluster mode, as etcd is an IO-heavy process, resulting in poor performance on IoT devices with memory card hard drives. However, in this paper, the focus is on edge computing and since there are no constraints on IO performance, the more mature etcd is used.

To enable etcd the `K3S_DATASTORE_ENDPOINT` environment variable needs to store the URLs of the etcd cluster, besides the `CAFILE`, `CERTFILE` and `KEYFILE` variables rreference the according TLS certificates of the already running etcd cluster. For multi-nodes setups, the environment variable `K3S_TOKEN` has to be set and used for adding additional nodes.

After providing the necessary etcd and secret variables, there is only the need to disable traefik as the default ingress-controller. Since the ODS does not use traefik, its installation is disabled by passing `-no-deploy traefik` to the `INSTALL_K3S_EXEC` variable. By default, K3's primary nodes also act as worker

nodes to remain as Elastic as possible. This behavior can be prevented by adding a taint to the primary node.

Automatization Scripts

It is possible to automate all of the steps above. Therefore, the *infra* folder of the ODS-deployment repository is used. Installing the etcd cluster and k3s on dedicated primary nodes is handled in the *setup_initial_cluster.sh* script, expecting an odd number of targeted machine IPs, with root access and an apt-based installer.

1. Checking installed tools, previous configurations and assumptions
2. Generate a root certificate and persist it into a local *CERTS* directory
3. Generate and persist *mem ber-$index* certificate and key files
4. Copy the certificate chain with a hardcoded path into the */etc/etcd/ssl/* directory on target.
5. Execute *utils/install_etcd.sh* script on the target machine:
   (a) Transform all target IPs into etcd peer URLs
   (b) Download the etcd binary and verify the installation
   (c) Create data directory and grant user permissions
   (d) Concatinate configuration passed as script argument with the hard-coded TLS certificate configuration
   (e) Create a system service, linking the created binary and configuration file
6. Start the system services and wait until the etcd cluster is up and healthy.
7. Create and persists a cluster *TOKEN* file inside the *CERTS* folder
8. Execute *utils/install_k3s_primary.sh* script on the target machine:
   (a) Transform all target IPs into k3s etcd URLs
   (b) Execute fetched k3s install script
   (c) Taint primary node with *NoSchedule*

In industry, executing automated scripts on targets machines is done with Ansible instead of piping scripts into the ssh command. However, in step 2.b, the *install_etcd.sh* script expects the cluster configuration as shell parameter. This parameter is dynamically generated for either etcd cluster-setup or an etcd

member-add operation and Ansible does not provide support for dynamic parameters. Also, the installation scripts are small, so Ansible is omitted as an external dependency.

There are also scripts for adding a primary and a worker node and for updating the k3s versions. For these add operations, the TLS chain and the token file from the *CERTS* folder must be installed on the local computer. These scripts are the stored output from the commands above and are not explicitly explained.

Furthermore, there are scripts for removing nodes from a cluster; this is especially necessary because an etcd cluster with an unreachable node is marked unhealthy and immutable. Uninstalling k3s on a machine and removing the node from a cluster, on the other hand, is a trivial task.

Infrastructure as Code

In the Kubernetes world, everything can be stated declarative. This also includes hardware and tools like Terraform are used broadly to allocate servers and define clusters with code. Since Kubernetes includes the ClusterAPI, which recently left alpha state, this was the chosen provider for IaC. The ClusterAPI is a plain interface and the implementation heavily relies on the cloud provider. Amazon, Microsoft, Redhat and every other major cloud provider offer their individual contract implementation for their environment.
Consequently, there is no leading implementation for bare-metal environments. Metal3 is an OpenStack-based operator and according to the homepage: 'It enables the provisioning and management of bare-metal machines throughout their lifecycle.' Nevertheless, this work is based on VMs, not bare-metal servers and Metal3 requires Intelligent Platform Management Interface (IPMI) from supported hardware. It is possible to simulate IPMI on VMs, but the documentation is incomplete and the example provided with 4 VMs did not work on a Fedora 34 machine.

Metal-As-A-Service (MAAS) is another service provided by Canonical for remote control of bare-metal servers with a ClusterAPI provider. Nonetheless, installing and configuring MAAS on the target servers requires more time, configuration and dependencies than setting up the K3s cluster, so this solution was also discarded.

Since there is no other mature OSS implementation for bare-metal environments, ClusterAPI is not used in this paper. There is also no need to configure the small number of VMs via ClusterAPI or any other infrastructure-as-code tool. Furthermore, this work provides all the necessary tools for adding or removing an appropriate number of nodes, leading to minimal and atomic maintenance tasks.

5.3.3 Hybrid cluster

A hybrid cloud is defined as a combination of private and public cloud resources. Hybrid cloud configurations are used either to bundle functions, e.g., a database cluster on AWS or to achieve peak load capacity by offloading the workload to external VMs (Zhu et al., 2018).

Kubernetes proposes the Cloud Controller Manager (CCM) to obtain and manage nodes. This API allows different cloud providers to integrate their platforms with the orchestrator, by controlling nodes, routes and services. Due to the CCM is the reason Kubernetes clusters of different vendors cannot simply be joined, as every vendor uses its own CCM plugin. However, k3s brings its own CCM implementation, which is only tied to the installed OS.

Since the elasticity concept of this work is built on the characteristics of horizontal scaling, there is no automatic allocation of cloud resources. Nevertheless, there is a proof of concept for AWS instances as the installation steps are the same as for a bare-metal VM, with only slightly changes on the ssh credentials.

```
infra/add_agent_node.sh  \
    "131.188.64.171" "3.94.79.171" \
    "AWS" "./AWSkeypair.pem"
```

Figure 5.6: Install k3s on AWS

The command of figure 5.6, adds a EC-2 instance to the existing cluster at 131.188.64.171. The public IP '3.94.79.171', refers to an AWS t2.micro with Debian 10 and open ports on 22, 80, 443 and 6443. The 'AWSkeypair.pem' parameter is the path to keypair of the instance initiated by the AWS Dashboard.

Adding and removing primary nodes from AWS is not supported, as controller nodes should remain within the cluster permanently, and the script is restricted to agent nodes.

5.4 Cluster ecosystem

Since Kubernetes is a platform to build platforms, an ecosystem needs to be established. Every ecosystem component of table 5.1 has multiple possible implementations. In the following, each component is evaluated, discussed and installed. The installation is performed all-in-one by applying a local Helm chart.

Name	Description	CCC
Metrics	Provide metric server, for scraping and storing data	Yes
Longhorn	Replicated storage and storage monitoring	Yes
Certificates	Retrieve and renew TLS certificates for webservices	Yes
Service-Mesh	Service-Mesh implementation for traffic tracement in distributed systems	Yes
GitOps	Automatic reconciliation of the infrastructure code from a Git repository with the cluster state	Yes
PostgreSQL	PostgreSQL Database in high availabiliy mode	No
RabbitMQ	RabbitMQ Broker in high availabiliy mode	No

Table 5.1: Cluster operators

Operators are either used for solving cross-cutting concerns and located at the lower layers of the IT stack or installed as a custom resource that solves a more general task. Either way, those resources need to be installed with care, as a deployed operator usually runs in HA mode and is written in Golang, resulting in not neglectable resource demand. Furthermore, not tracking installed operators can result in a bloated and inefficient cluster or in Shadow IT.
Shadow IT is a problem in big projects when developers implement workarounds, services or even infrastructure. Kubernetes clusters are also prone to this problem (Haag & Eckhardt, 2017).

To not lose track of installed operators, the */helm/operator* centralizes the installation of all Operators, and to prevent Shadow IT, this chart needs to be applied by a cluster administrator and has no pull based sync automatization.

In the following, only the cross-cutting features are elaborated, while more specific operators will be explained in detail for the actual ODS implementation.

5.4.1 Monitoring

Monitoring a cluster is as essential as deploying it. Displaying resource usage and disk utilization and issuing alerts are essential for developers and administrators. However, humans and the HPA need a metric instance through which resource utilization can be retrieved. Consequently, installing a metrics server is necessary for introducing an Elasticity concept.

Prometheus is an OSS time-series database and uses this data source to generate alerts. The CNCF Foundation keeps this project, which was started by Google, in a tiered state, tagging Prometheus as a first-class metrics server. Traditionally, Prometheus is used along with Grafana, a web UI focusing on customizable and powerful dashboards. A Prometheus alert can be compelling, from triggering when a Pod is down for a fixed amout of time, up to a estimation, when the disk will be filled. Sending a triggered alert is done by the AlertManager, scraping Prometheus errors and broadcasting it to configurated receivers.

Setting this all up is a standard task for each fresh Kubernetes cluster, resulting in a preconfigured package. Installing *prometheus-community/kube-prometheus-stack* via Helm automatically installs all above components, adds Prometheus alerts for every kind of cluster malfunctions and configures Grafana dashboards.

The only thing, which needs further customization is the AlertManager. The easiest way to send alert emails is by reapplying the external Helm chart but with additional user variables, including sender SMTP configuration and credentials and at least one receiver email. After applying the parameterized chart, the credentials are stored within a cluster secret and the value file containing the cleartext credentials has to be deleted. A misconfigured the AlertManager fails silently, to ensure reliable email alerts, the script *utils/setup_alert_emails.sh* performs the above steps in a cli-typical manner and can be highly recommended to use.

5.4.2 Network file system

Kubernetes supports different types of files systems, for example: AzureFiles, VsphereVolume and Local. Like traditional files systems each of the StorageClasses, serves a specific purpose: AzureFiles provides a Network File System (NFS) on the Azure cloud, while VsphereVolume can mount a local Virtual Machine Disk File on the cluster.

The default local file system works out of the box, but is difficult to monitor and does not support replication or backup of the stateful container volumes. In addition, each Pod must be scheduled with a PersitenceClaim on the node where the volume resides. Adding some more - not always resolvable - constraints to the scheduler affecting the QoS of the entire application. During this work, all major incidents were due to disk pressure, resulting in data loss, inconstant application state, and in some cases lead to unrecoverable total cluster failure.

The CNCF supports the *Longhorn* project in incubating state, this open source NFS, has a built in web dashboard for monitoring the summarized cluster file system state. As shown in figure 5.7, Longhorn support replicating, incremental snapshots and backups. Hence it is used as the default StorageClass, for k3s

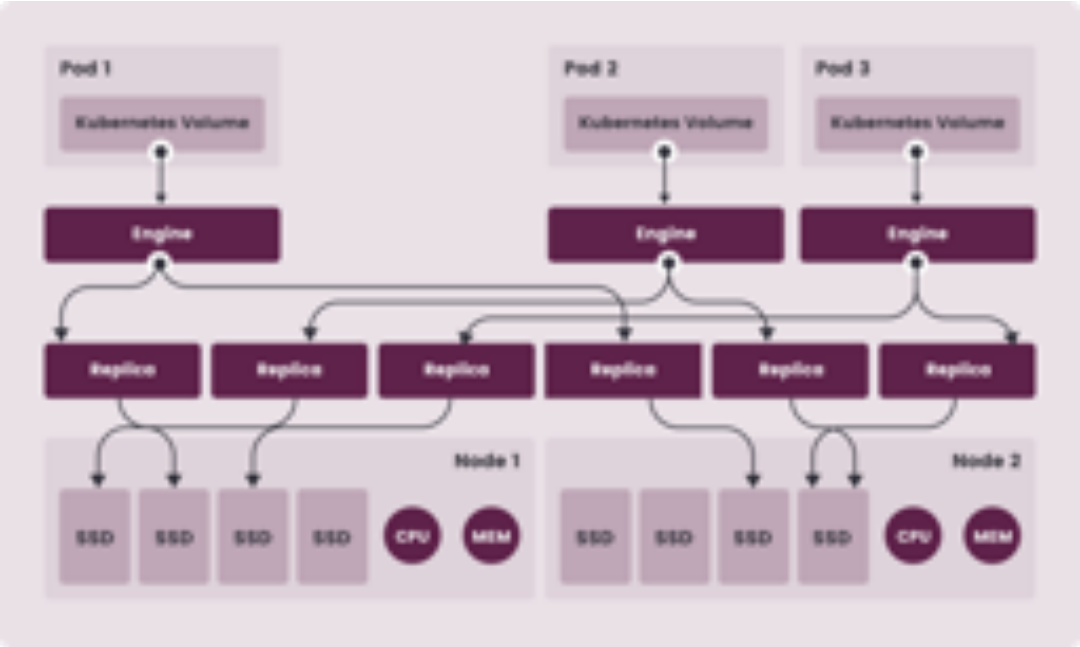


Figure 5.7: Longhorn architecture (Longhorn.io)

based setups. The disadvantage is that Longhorn adds a native dependency and *open-iscsi* must be installed on the host operating system.

5.4.3 GitOps

ArgoCD and Flux are two GitOps tools in the CNFC incubated state. Flux has no official web interface and, thus, is dropped in favor of ArgoCD. Argo v1.0 was released in Q1 of 2019 and is mainly developed by Akuity, Intuit and RedHat employees. After applying the operator, ArgoCD is ready to use and automatically deploy workloads within the cluster.

Bootstrapping

To deploy a Helm chart from a Git repository, ArgoCD expects a configurated Application. For a better overview and rights management, Applications are bundled into AppProjects. In addition, an AppProject expects several Git repositories from which it can be retrieved as well as the namespaces in which the associated Applications can be deployed. The application resource needs the repository URL, the git revision and the file path of the remote Helm chart. ArgoCD is then ready to reconcile the cluster state with the declarative state inside the Git repository.

The detailed structure of the ODS Helm charts is explained in chapter 5.5.4; for now, every microservice resides in its individual chart and is deployed in

Figure 5.8: ArgoCD UI

its own Argo Application. Each Application has a sync state and one of several health states. The sync state indicates if the desired manifest from Git equals the deployed manifest and the health state specifies the component's fitness. Figure 5.8 shows a screenshot of a deployed ArgoCD service. The orange icons displays an out-of-sync issue that cannot be reconciled, but the service itself is still in a healthy state. Not only can an application be displayed, but it is also possible to create, delete Applications, live YAML modification or retrieve logs from Pods, resulting in a web UI as powerful as `kubectl` and the Lens IDE.

Workflow

The GitOps workflow is quite similar to the Git-Workflow:

1. A developer takes a ticket to implement a task, e.g., migrate a new Service to the cluster
2. The main branch is forked in order to create a feature branch
3. The developer implements the feature
4. The patch is published within a Pull Request into the main branch
5. A CI pipeline checks for a valid Helm syntax with `helm template`
6. A reviewer discusses possible improvements the developer might implement
7. The reviewer approves the Pull Request
8. The developer merges the branch into the main

The only difference to the ODS Git workflow is that there is no triggered CI pipeline for deployment right after the merge. ArgoCD will pull the state from Git and apply it to the cluster. Programming infrastructure is a risky task as even minor syntax errors might freeze a cluster, delete a production database or make ArgoCD inaccessible. Consequently, the review process is an essential part of this workflow.

Testing the provided infrastructure patches is sometimes limited to the local machine, for instance, the automated retrieval of TLS certificates, which needs to be done with an external URL. Helm has a lint subcommand to validate the correctness of templates, but that is not checking Kubernetes specific constraints. Dry-run deployments are also not possible on the productive cluster, as Kubernetes handles duplicated resources the same way as a malformed YAML.

Monitoring deployments

ArgoCD deploys and deletes components automatically. Despite the review process described above, a deployment may fail due to a bad patch or a general incident of the ODS services may appear, leaving the deployment in an unhealthy state. As already stated, Argo supports several health states out of the box:

- **Healthy**: the Application is deployed and live/health checks passed
- **Progressing**: the Application is currently reconciled
- **Degraded**: the deployed Application is down or live/health checks failed
- **Subspended**: reconciling is stopped
- **Missing**: the component cannot be installed
- **Unknown**: unknown error

Vanilla ArgoCD does not support sending and receiving alerts on state changes. The external plugin Argocd-notifications adds this functionality by providing so called senders for Slack, email and others. To bootstrap the plugin, each Application must be annotated with a trigger, a sender and a recipient. Registering a sender is nontrivial as it is necessary to edit the plugins configmap, but sadly this configmap is misused by not using an array or a map, but a string field, where the config is stored as individual JSON, split per line. Furthermore, the configmap data string field holds all other plugin data, text templates and the definition of when to trigger a notification. Modifying this apparent data structure needs to be done with patch commands:

```
kubectl patch cm argocd-notifications-cm -n argocd --type merge -p \
"{\"data\": {\"service.slack\": \"token: \$my-slack-token\nicon:"}}"
```

For security reasons, the {{"token": "my-slack-token" }} JSON field is a reference to a hard-coded secret where my-slack-token is stored. This configmap is also configured via a `my-slack-token: $ACTUAL_TOKEN` line inside a string field. Adding an email sender follows a similar but slightly tweaked and even more complicated variant of this. Adding a recipient is comparatively simple, which is the value of the annotation: `notifications.argoproj.io/subscribe.slack=$CHANNEL` This can be tweaked further by adding a trigger like "on-health-degraded" before the sender. Without an explicit trigger name, all default triggers are fired.

Besides this plugin's more than questionable design, the notifications themselves work reliably, allowing the receivial of granular updates on the clusters deployment and error state.

Latest image deployment

Deploying a pod in Kubernetes is done by deploying a container image. For Helm, it is best practice to parameterize this image by specifying a `image.name` and `image.tag`:

```
image: "{{ .Values.image.name }}:{{ .Values.image.tag }}"
imagePullPolicy: "{{ .Values.image.pullPolicy }}"
```

The imagePullPolicy being set to "AlwaysPull" and with image tag set to 'latest', one might derive that this will automatically pull the latest image, but that is not the actual behavior. Each time a Pod gets started, the latest image might be pulled and it is not deterministic when a the fresh Pod gets started. For instance, the Pod might crash or the HPA scales up and forces of a new creation of a Pod, resulting in two different running images. To avoid this problem, using the 'latest' tag is discouraged and a fixed tag is used, enforcing a stable container on all deployed pods.

To update a container, manually updating the Helm Chart's tag field is required. Kubernetes then performs a rolling release and shuts down the old pods while starting the updated image. This might be an acceptable solution for production environments, as rolling out the productive release is not an everyday task. For integration and other development environments, an automatic CI/CD is a must.

```
$ argocd-image-updater test \
   ghcr.io/jvalue/open-data-service/ui \
   --update-strategy latest \
   --ignore-tags latest \
getting image
found 7 from 7 tags eligible for
    consideration image=ghcr.io/jvalue/open-data-service/ui
latest image ghcr.io/jvalue/open-data-service/ui:0.1.0-ef999cb
```

The Argo CD Image Updater is a plugin that automates the manual step of updating the Helm tag parameter to the *latest pushed* image tag. Hence the Image Updater needs the image registry path and update strategy. The returned latest tag might actually be 'latest' - for avoiding going full circle, this tag needs to be ignored.

Quite similar to the notification plugin, the updater is enabled via annotations on the Argo Application resource. As there might be more than a single image in a chart, the corresponding Helm parameter also has to be set:

```
argocd-image-updater.argoproj.io/image-list="$id$=$image"
argocd-image-updater.argoproj.io/$id.update-strategy=latest
argocd-image-updater.argoproj.io/$id.helm.image-name=$id.image.name
argocd-image-updater.argoproj.io/$id.helm.image-tag=$id.image.tag
argocd-image-updater.argoproj.io/$id.ignore-tags=latest
```

Configuring the image repositories, Helm parameters and constraints is not a generic task. Every chart has its unique values and constraints and the actual usage of this plugin for the ODS is done in chapter 5.5.4.

5.4.4 Service mesh

Istio and Linkerd are public Service-mesh implementations. Since the Linkerd version tested did not work properly with RabbitMQ traffic, which is at the heart of the ODS, and Istio offers more extensive plugin support, Istio is chosen as the service mesh implementation.
Istio is the only operator which does not provide a simple operator install via `kubectl apply`. Instead, the `istioctl` binary is shipped and performs the installation. To visualize the service mesh UI, a Prometheus instance needs to be configured alongside Istio.

The pods' namespaces need to be annotated to enable automatic sidecar injection. After the annotation, each started Pod gets its sidecar injected, and the traffic can be shown via Isitios web interface (kiali).

Istio Gatway

In order to track external traffic and map it into the Istio traffic, Istio needs to capture the external traffic. Ingress usually does map external to cluster traffic, and Istio would be the Istio-controller instance. But Istio bypasses this approach by introducing VirtualServices and Gateways.
Gateways allocate ports by defining the protocol, so port 80 is reserved for HTTP connections and port 443 for HTTPS. TLS connections need additional configuration with certificates, which is explained in chapter 5.4.5. Additionally, each Gateway needs at least one host URL. There can be N Gateways, with M Hosts,

where $M >= N$, resulting in calls to 'https://elastic-ods.tk' be forwarded to a different Gateway than a call to 'https://integration.elastic-ods.tk'.

Each virtual service has exactly one Gateway and acts as a rules-based reverse-proxy, just like Ingress. For instance, a VirtualService rule is to match the prefix: '/API/adapter/' and forward the request to the adapter-service on port 8080. This results in an external call 'https://elastic-ods.tk/api/adapter/' that would match a Gateway's host and port and then check for any match inside the Gateways linked VirtualService, finally applying the defined user rule.

Due to the port allocations of Istio, vanilla Ingress is completely replaced by Istios Gateways and VirtualServices, but as they serve the same purposes, they are used as synonyms.

Figure 5.9: Displayed ODS traffic with kiali

After correct initialization, all traffic is fully traceable and TLS-secured. Figure 5.9 shows kiali, which can be used to view TCP, gRPC and HTTP traffic and analyze response times, traffic distributions and success rates. Istio ultimately enables benchmarking and error tracking for all applications running in the cluster.

5.4.5 TLS certificates

Encrypted TLS traffic is a must for any website, enforced mainly by modern browsers. Globally trusted certificate authorities like 'Let's Encrypt' with their support of an Automatic Certificate Management Environment (ACME) enables the programmatical retrieval of certificates. The Cert-Manager is a Kubernetes operator designed for automated AMCE certificate retrieval.

1. Each Certificate has one Issuer, the Issuer resource then needs an email and the AMCE server address
2. The Certificate is linked to the Issuer and configures the DNS and the commonName
3. The Cert-Manager creates an Order for Certificate retrieval
4. The Order creates a CertificateRequest
5. The CertificateRequest creates:
   (a) An Ingress-Endpoint for the generated ACME query
   (b) A Pod to resolve the ACME query
6. Back to step 1, when the Certificate is about to expire

However, this naïve approach means that the traditional AMCE steps are translated from a bare-metal server to Kubernetes resources, but Kubernetes has additional network layers that need to be further configured: firstly, Istio replaced Ingress and blocks the HTTPS port and the endpoint slice for step 5a, hence the Istio controller needs to be set as the global Ingress controller.
Second of all, these steps must be performed on the production cluster, as the external URL of 5b must return the correct AMCE challenge.
Last but not least, HTTPS communication in 5b is impossible without a TLS certificate, but Cert-Manger does not generate a self-signed certificate by default. Therefore, the certificate resource needs an additional annotation.

The 'Let's Encrypt' staging environment is used by default to fix the above issues and enable local development while not running into any quota. Nevertheless, standard browsers do not accept a certificate signed by the staging authority. ArgoCD has the option to change the Helm parameters inside the web UI, but setting this variable alone will not trigger the Cert-Manager to create a new Order. The rewnewBefore parameter must also be set to five minutes after the initial Certificate generation and reset right after. However, this needs to be done only for the initial retrieval.

5.4.6 Autoscaler configuration

The HorizontalPodAutoscaler is the essential Kubernetes Resource for Elasticity of any kind. To configure the HPA, one must specify metrics with the desired utilization value and a lower replica limit; the upper limit is optional. Afterward, the current replica count is continuously adjusted based on formula 5.1 - adapted from the official Kubernetes documentation.

$$desiredReplicas = min(max(\arg\max_{\theta} f(x,y), minReplicas), maxReplicas)$$
$$f(currentMetricValue, desiredMetricValue) = \lceil currentReplicas * currentMetricValue/desiredMetricValue \rceil \quad (5.1)$$

Where θ consists of tuples of the current and desired metrics values. The variables $minReplicas$, $maxReplicas$ and each $desiredVal$ are hyperparameters, while $currentMetricValue$ is a metric derived from the observable system state.
The variable $currentReplicas$ acts as a multiplier in the formula; its initial value is the lower replication limit. Thus the lower bound is an important variable for the overall elasticity.
The used metrics in this thesis restrict to compressible resources, more specifically CPU and primary memory. But since HPAv2beta2 support custom metrics, values like the number of HTTP connections, are also possible. As custom metrics are very concrete for the service implementation, they are not considered in the general Elasticity concept.

As demonstrated in chapter 6.3, a minimum replica count of *5* and an average metric utilization of *80%* achieve the best balance between being Elastic and not overusing cluster resources.
To scale the microservices down, the HPA recalculates the desired replica count of the averaged last 60 seconds but is constrained to remove only 50% of the available pods. Hence the system is not overprovisioning for longer than one minute but still stays responsive during sudden load peaks.

5.5 ODS architecture

After bootstrapping Kubernetes and setting up an ecosystem, the ODS is ready to be deployed. The general Helm architecture is shown in figure 5.10. There are sub-charts for the database, Ingress, RabbitMQ and the microservices. The microservice chart consists of multiple sub charts, explained in chapter 5.5.4. This top-down architecture allows for a more granular installation of the ODS as one can either install the entire ODS with a single Helm command or the individual sub chart - resulting in multiple advantages:

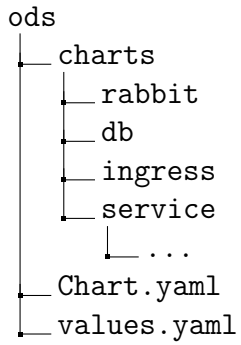


Figure 5.10: ODS Helm chart

- Local Development: applying only needed services safes up resources and startup time
- GitOps Deployment: to not accidentally delete/modify stateful middleware resources
- CI pipeline: single `helm lint` command for the whole infrastructure code

List 5.11: ODS sub chart architecture advantages

In the following chapters explain the components:
Starting with the usage of the different ArgoCD value files for different deployment environments, followed by the non-CCC operators (PostgreSQL and RabbitMQ), the ODS service installation and finally the service exposure via Ingress.

5.5.1 Integration and production stage

The ODS already has a Continous Integration (CI) pipeline, nevertheless with this chapter, the pipeline is extended into a Continous Deliverment (CD) pipeline, by introducing an integration and a production environment.

The integration stage acts as a test environment, in which the latest state of each ODS service is automatically deployed. The result is an environment that can be used for manual or end-to-end testing in a production-like environment. Furthermore, an additional quality step is added as each developer can test their patches on a target system without the risk of corrupting the production ODS instance. This also applies to infrastructure-related code that directly affects Resilience and Elasticity (Arachchi & Perera, 2018).
The production stage is the live ODS using fixed Container-tags, guaranteeing a tested and stable application.

Setting up ArgoCD

The ODS is deployed via ArgoCD Helm charts. Since Helm supports multiple values files Argo is also supporting them. Hence every Helm chart inside the ODS has a `values.yaml` and a `values-integration.yaml`. The production values include HPA and replication with a fixed image, while the integration values have the latest image and only one fixed replica.

Furthermore, to automatically pull the latest Image Tag, the ArgoCD-Image-Updater needs to be setup properly. Therefore, the Helm 'image.updater' template is defined. This template expects a tuple as parameter, the first tuple value is the Helm root variable, for accessing the `$root.Release.Namespace` and the full image URL, which shall be updated.

```
{{- $root := index . 0 }}
{{- $image := index . 1 }}
{{- $id := $image.name |
  trimPrefix "ghcr.io/jvalue/" |
  trimPrefix "open-data-service/" |
  replace "/" "-" }}
```

Finally, a CronJob is created to set or delete the individual image annotations of the Argocd AppProject. For that reason, a `kubectl` container is run, checking if the container tag is equal to *latest* or has a fixed image tag, after the check the script adds or removes the necessary annotations. The container has addtional RBAC configuration (`["get", "list", "annotate", "patch"]`) for performing the annotations on the AppProjects and runs everyday at midnight.

5.5.2 RabbitMQ Cluster

The MOM of the ODS is RabbitMQ, for HA cluster, the operator from the official website is used. This allows an easy setup of RabbitMQ inside Kubernetes.
The operator takes a moment to instantiate, but most of the ODS services would crash without an instance already running. Thus every service startup is delayed due the `initContainers` field, blocking until the command in figure 5.12 succeeds.

To authentificate with the broker, the credentials from a generated secret need to be pulled. To avoid repeating boilerplate code, a template like shown in figure 5.13 is written and included in every dependent service.

5.5.3 Database Cluster

Modern REST APIs are inherently stateless and idempotent, meaning the services themselves do not store any data. Consequently, an API call can be made

```
until wget http://$(RABBIT_USR):$(RABBIT_PWD) \
    @$(RABBIT_HOST):$(RABBIT_MGMT_PORT) \
    /api/aliveness-test/%2F
do
  echo "waiting for RabbitMQ startup";
  sleep 2;
done;
```

Figure 5.12: RabbitMQ await template

```
{{- define "rabbitmq.env" -}}
- name: RABBIT_USR
  valueFrom:
    secretKeyRef:
      name: "ods-rabbitmq-cluster-service-default-user"
      key: username
- name: RABBIT_PWD
  valueFrom:
    secretKeyRef:
      name: "ods-rabbitmq-cluster-service-default-user"
      key: password
- name: RABBIT_HOST
  valueFrom:
    configMapKeyRef:
      name: rabbitmqt-config
      key: hostname
{{- end -}}
```

Figure 5.13: RabbitMQ environment variable template

multiple times without changing the result. Therefore, databases store the data processed by the business layer services. Since the Open Data Service already has data in its name, proficient data storage is mandatory.
The ODS is built on top of PostgreSQL (Postgres), a high-performance, open-source and relational database founded in 1996. PostgreSQL is a popular Database, and for this reason, a lot of Kubernetes operators exist. Those operators typically enable a Postgres cluster in HA mode, with a leader and read-only replicas.

Zalando's 'Postgres Operator' is one of the most actively developed operators and is backed by a company that relies on it. Moreover, the ODS outboxer relies on a Postgres native feature, namely *Logical Decoding* to extract the database changes. Logical Decoding relies on Write-Ahead Logging (WAL); for the outboxer, the `wal_level` needs to be to the highest level: `logical`. Most of the Postgres Operators replicate due to `wal_level = replica` and changing the parameter

disables HA replication. Making this the only tested operator, that meets all requirements.

Integration to ODS

For every ODS environment, there is a central PostgreSQL instance installed. Each service has its own Postgres database and user within this instance. This centralized approach was used instead of database sharding because there was no bottleneck and a centralized approach is easier to maintain.

Retrieving the Database credentials is done via an operator generated secret. As accessing the secret is a repeative task, this is wrapped into the 'postgres.env' template, similiar to figure 5.13. Some ODS microservices initially haven't been able to communicate to the Kubernetes Postgres instance, due to its self-signed certificates. So a patch is provided to enable these certificates via the `POSTGRES_SSL` environment variable.
Although the Postgres Cluster has a short startup time, there is an `initContainers` template for awaiting the DB to be reachable without crashing the container. Thus the 'postgres.await' - similar to figure 5.12 - loops until the `psql` CLI successfully connects to the Postgres instance.

5.5.4 Services

The service chart serves as the parent chart for the individual sub chart, where each sub chart represents an ODS microservice. All services are listed in figure 5.14 and are explained in this chapter one by one.
This top-down design and the granular deployment options serve the same benefits as already shown in list 5.11, with a particular emphasis on easier local development. However, the implementation of the service relies on middleware, the deployment is coupled with the configuration config files and the Postgres and RabbitMQ charts have to be deployed before any services.

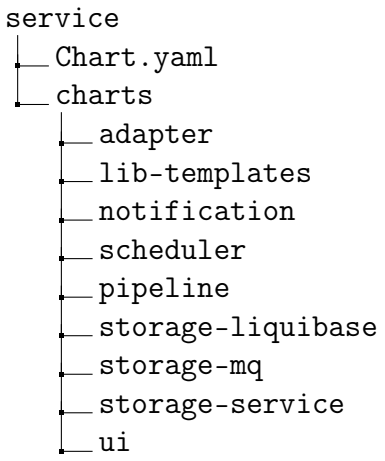


Figure 5.14: ODS service chart

All Microservice charts are summarized in tables as they follow the service/deployment pattern of appendix A. Only the image, minor configuration and the environment variables differ, which are reflected in the `values(-prod).yaml`.

The only exception is the lib-template chart, a *lib-templates* local Helm library, for sharing general templates over the ODS service charts. Collected boilerplate

templates, which need only small to none parameters, reside in this package and will be explained briefly below:

- **_image_updater.tpl**

  'image.updater' expects the chart root and the image to update parameters to configure the ArgoCD automatic image updater, by deploying a CronJob.

- **_outboxer_deployment.tpl**

  'outboxer.deployment' expects the chart root to retrieve the database name and then deploys a single outboxer instance. The outboxer is waiting with 'postgres.await' and 'rabbit.await' for the environment to start-up and authenticates with 'rabbitmq.env' and 'postgres.env' parameters.

- **_hpa.tpl**

  'hpa.config' expects the chart root and a service-name as an optional parameter.
  The template retrieves the CPU and memory thresholds (80) and the minimal replica count (5) as the config parameters from the chart root values.

- **_postgres_await.tpl**

  'postgres.await' blocks until there was a successful `psql` connection on the chart database.

- **_postgres_env.tpl**

  'postgres.env' reads the secrets and configmap of the dependent PostgreSQL chart and saves it into environment variables. This is purely done to reduce boilerplate

- **_postgres_connecton_test.tpl**

  'postgres.connection-test', is for local development tests, and indicates if a `psql` call to the configurated database was successful.

- **_rabbitmq_await.tpl**

  'rabbit.await' blocks until there was a successful `curl` on the health endpoint of the RabbitMQ broker.

- **_rabbit_env.tpl**

  'rabbitmq.env' reads the secrets and configmap of the dependent RabbitMQ chart and saves it into environment variables, also to reduce boilerplate.

These templates are used to deploy the ODS services. As the ODS is not particularly important for the Elasticity concept, there is only a brief explanation of the used Kubernetes resources.

Adapter Service

ghcr.io/jvalue/open-data-service/adapter		
templates ├── outboxer.yaml ├── image-updater.yaml ├── deployment.yaml ├── hpa-config.yaml └── tests └── postgres.yaml	initContainers	**- postgres.await** **- rabbitmq.await**
	DatabaseId	**adapter**
	Outboxer	**Yes**
	Port	**8080**
Horizontal Pod Autoscaler	**2-5 Replicas on production** **- targetCPUUtilization: 80%** **- targetMemoryUtilization: 90%**	
Image-Updater	**On integration**	

Notification Service

ghcr.io/jvalue/open-data-service/notification		
templates ├── hpa.yaml ├── image-updater.yaml ├── deployment.yaml └── tests └── postgres.yaml	initContainers	**- postgres.await** **- rabbitmq.await**
	DatabaseId	**notification**
	Outboxer	**No**
	Port	**8080**
	limits	**80 Mi**
Horizontal Pod Autoscaler	**2-5 Replicas on production** **- targetCPUUtilization: 80%** **- targetMemoryUtilization: 90%**	
Image-Updater	**On integration**	

Pipeline Service

ghcr.io/jvalue/open-data-service/pipeline		
templates ├── hpa.yaml ├── image-updater.yaml ├── outboxer.yaml ├── deployment.yaml └── tests └── postgres.yaml	initContainers	**- postgres.await** **- rabbitmq.await**
	DatabaseId	**pipeline**
	Outboxer	**Yes**
	Port	**8080**
Horizontal Pod Autoscaler	**2-5 Replicas on production** **- targetCPUUtilization: 80%** **- targetMemoryUtilization: 90%**	
Image-Updater	**On integration**	

Scheduler Service

ghcr.io/jvalue/open-data-service/scheduler		
templates ├── image-updater.yaml ├── outboxer.yaml └── deployment.yaml	initContainers	**- postgres.await** **- rabbitmq.await**
	DatabaseId	**scheduler**
	Outboxer	**Yes**
	Port	**8080**
Horizontal Pod Autoscaler	**No, singleton service**	
Image-Updater	**On integration**	

Storage Service

ghcr.io/jvalue/open-data-service/storage-db-liquibase		
templates └── liquibase-job.yaml	initContainers	**- postgres.await**
	DatabaseId	**storage**
	Outboxer	**None**
	Port	**None**
Horizontal Pod Autoscaler	**None, AlwaysPull policy is sufficient**	
Image-Updater	**On integration**	

ghcr.io/jvalue/open-data-service/storage-mq		
templates ├── deployment.yaml ├── image-updater.yaml └── hpa.yaml	initContainers	**- postgres.await**
	DatabaseId	**storage**
	Outboxer	**No**
	Port	**8080**
Horizontal Pod Autoscaler	**2-5 Replicas on production** **- targetCPUUtilization: 80%** **- targetMemoryUtilization: 90%**	
Image-Updater	**On integration**	

ghcr.io/jvalue/open-data-service/storage		
templates ├── deployment.yaml ├── image-updater.yaml ├── hpa.yaml └── tests └── postgres.yaml	initContainers	**- postgres.await**
	DatabaseId	**storage**
	Outboxer	**No**
	Port	**3000**
Horizontal Pod Autoscaler	**2-5 Replicas on production** **- targetCPUUtilization: 80%** **- targetMemoryUtilization: 90%**	
Image-Updater	**On integration**	

UI Service

ghcr.io/jvalue/open-data-service/ui		
templates ├ deployment.yaml └ image-updater.yaml	initContainers	**None**
	DatabaseId	**None**
	Outboxer	**Yes**
	Port	**80**
Horizontal Pod Autoscaler	**2-5 Replicas on production** **- targetCPUUtilization: 80%** **- targetMemoryUtilization: 90%**	
Image-Updater	**On integration**	

5.5.5 Bootstrapping ArgoCD

The ODS Helm charts are ready to use and can be deployed manually. ArgoCD is also already present on the target cluster. To finally bootstrap the ODS in ArgoCD, an additional Helm chart is required.

In this `argocd` chart, the AppProject and the Applications for *integration* and *production* are created. As each ODS Helm chart gets mapped into an application, the granularity of the deployed parent and sub-charts is defined here. The chosen granularity is 1:1, so every microservice gets its individual Application, resulting in 11 Applications for each ODS environment. The YAML template generation iterates over a list of the chart file paths of the according Git repository.

Consequently, this chart is not applied by default and must be deployed in Argo. The *bootstrap/bootstrap_gitops.sh* script solves this chicken or egg problem. The bootstrap script applies an *ods-bootstrap* named AppProject and Application, deploying the above *argocd* chart and making it a single point of truth. Once deployed, the *ods-bootstrap* Application can be deleted in non-cascading mode inside the ArgoCD web UI.

5.5.6 External access

All services and infrastructure are now deployed, ready to use and automatically synced via the GitOps workflow. To bring the ODS live, external access via a public domain needs to be setup.

Domain and DNS

Retrieving a official domain was no possible for this thesis, in addition the domain is hard to debug and it is difficult to implement the necessary changes within the

DNS provider. To speed up the launch of ODS, the free domain `elastic-ods.tk` will be rented for one year. The DNS provider for this domain will be changed to Cloudflare, as Cloudflare offers a simple web UI for further DNS customization. Cloudflare offers a comprehensive toolset such as Denial of Service protection or automatic certificates, but none of these features will be used to avoid vendor lock-in.
The DNS configuration consists of a wildcard A record of all subdomains and a root record pointing to the Kubernetes primary-Node IPs.

This multi DNS setup was developed during this work. Previously, only a single DNS record was set and the mapped machine IP was shared by the cluster, resulting in a major incident as the entire cluster was unavailable. This workaround is implemented since a single point of failure is neither Elastic nor Resilient. An external load balancer must be enabled to resolve this issue entirely, but this is a paid feature with Cloudflare and most other DNS providers.

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: {{ .domain }}-service
  namespace: {{ .namespace }}
spec:
  hosts:
    - "{{ .domain }}.{{ $.Values.host }}"
  gateways:
    - "{{ $.Release.Namespace }}/infra-{{ .domain }}-tls-gateway"
  http:
    - match:
      route:
        - destination:
            host: "{{ .destination.host }}"
            port:
              number: {{ .destination.port }}
```

Figure 5.15: Parameterized Ingress for an ecosystem component

Ingress

In Kubernetes, Ingress is a unique resource. Ingress is the collection of knowledge about external and internal components. External components, like the domain, the cluster capabilities and certificates, are mapped onto the services paths, URL rewrites and subdomains. Therefore, Ingress is centralized by design and resides in its own sub-chart.

Ecosystem configuration

The exposure of Argo, Istio or Grafana is as essential as their establishment and must be exposed with Ingress. This task amounts to obtaining a certificate and assigning all routes to a single service. As this is repetitive, it is done via parameterised helm charts, as shown in figure 5.15.

It is important to note that this only must be set up on the production environment and not on the integration environment. Otherwise the certificate retrieval would constantly fail, and the Cert-Manger loops forever.

ODS configuration

The UI service will make HTTP calls to each service in a `api/$SERVICE_NAME/$PARAMS` URL scheme. However, the receiving services expect the call without the identifying prefix, which needs to be stripped/rewritten.

```
- match:
    - uri:
        prefix: "/api/adapter/"
    rewrite:
        uri: "/"
    route:
    - destination:
        host: adapter-service
        port:
            number: 8080
```

Figure 5.16: Ingress route with stripped URL prefix

A route like in figure 5.16 has to be written for all services but is not done via Helm templates, as these are concrete implementation details. Additionally, the UI service is an exception, as it has to be placed last, for handling all calls, which are not handled by a service - similar to a default clause in switch-case expressions. To check the correctness of the Ingress rules, the command `istioctl analyse -A` displays an error if a referenced service was not found.

6 Demonstration

Considering what already was established in chapter 4, no Elasticity concept can exist without a Resilience concept. This unification also runs through this chapter, where Resilience is demonstrated primarily through benchmark tests and Resilience through chaos engineering. Nevertheless, overarching use cases such as zero-downtime releases do not fit into either category.

The following demonstration tests are performed:

1. Demonstration of the JValue ODS in its different environments
2. Zero Downtime deployment
3. Stresstests for triggering Kubernetes HPA
4. Chaos Engineering

6.1 Stages

The ODS inside the deployed Kubernetes cluster is reachable in an integration stage at 'integration.elastic-ods.tk' and a productive environment at 'elastic-ods.tk'. The integration phase is configured to automatically pull the latest JValue images for testing purposes while the production phase has fixed Git tags to ensure a stable product. To demonstrate the successful deployment of both stages, we define a fixed happy path within the ODS UI and analyze the systems behaviour with Istio.

1. Create a Datasource
   (a) Enter Datasource name '$Environment Datasource'
   (b) Use the default Datasource
   (c) Enter meta data with default values
2. Create a Pipeline
   (a) Enter Pipeline name '$Environment Pipeline'

(b) Transform function adds a $Environment String field

(c) Enter meta data with default values

3. Trigger the created pipeline
4. Check for transformed data in the Pipeline storage

Figure 6.1: Istio traces of the ODS in the production and integration stage

After executing the happy path with a Chrome browser, the traced calls are shown in figure 6.1. The production stage in the upper half works appropriately while the integration environment on the right seems to have problems with the memory service. The error only occurs in the actively developed integration environment; the error can be investigated further with Istio without affecting the production user data. This example shows how the distinction between the environments affects the application grade.

6.2 Rolling Releases

To maintain zero downtime in rolling release, the orchestrator still forwards requests to service while being upgraded. The orchestrator first starts a new Pod with the new version and then terminates a running Pod with the old version, this is repeated until the Pod set only consists of updated versions.

To test this behavior within Kubernetes, a newer version of the notification service is deployed. This action is performed by updating the image tag inside the

Figure 6.2: Active rolling release in ArgoCD

Git repository on the main branch and observing the system's behavior via the ArgoCD UI. As shown in figure 6.2, ArgoCD fetches the Git changes and triggers a rolling deployment.

ArgoCD also supports blue-green deployments, in this model, the application is divided into old (blue) and new (green) pods; the orchestrator starts all necessary green pods and then redirects all traffic at once to the green pods. Blue-green deployments is a more resource-intensive release model and therefore not further considered (Redhat, 2019).

6.3 Elasticity concept

The Horizontal Autoscaler is the core component of the Elasticity concept. Even though the available working machine does not have enough resources to generate enough load trigger ODS scaling mechanism, and a synthetic load generation test is chosen. A Fibonacci Service is implemented with Quarkus native; the fib-service calculates the n-th Fibonacci number and is accessible via a REST interface. The service is deployed within the same Service/Deployment templates alongside the HPA configurations.

Locust is a load testing framework, which is fully scriptable and automatically generates usage statistics. To start Locust, the number of users and a spawn rate must be set - each user has a single task that runs between one and three seconds. In this particular task, a call is made to the fib-service where n is in a range of 10-40 and is selected randomly with a linear distribution. The load test has the shape of figure 6.3, and after each spawn, there is a five minute wait period:

- 100 Users, spawn rate 5
- 500 Users, spawn rate 5
- 750 Users, spawn rate 10
- 1500 Users, spawn rate 3
- 500 Users, spawn rate 5
- 100 Users, spawn rate 5
- 1 User, spawn rate 5

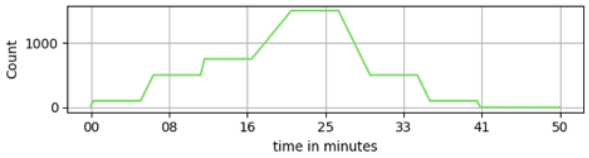


Figure 6.3: User load shape

The fib-service and the HPA resources are re-created before each test. A python script is written to measure the replica count; it watches for Pod action via the official Kubernetes API and filters the add and delete operations. The observed HPA variables are the minimal replica count (N) and the average CPU utilization (U). There is no maximal Pod count set because this would not affect the Elasticity. The individual Charts are in appendix B until E.

The ODS HPA configuration of chapter 5.4.6 is referring to this chart and the *currentReplicaCount* and the *metricUtilization* are the two variables considered in figure 6.4. In the figure, the utilisation factor defines an almost linear slope while the replication factor makes the graph slightly exponential.

For real-world applications, this means that the utilisation factor is more important for very volatile and inconstant workloads, although the minimum/current replica count is more important for more constant, but steep gradients. In the case of the expected workloads of the JValue ODS, this leads to a higher replica count and a also higher resource utilization factor. Furthermore, the allocated cluster is relatively small in its current state and low utilization would eventually lead to resource stealing for other services due to an too overprovisioning Elasticity concept.

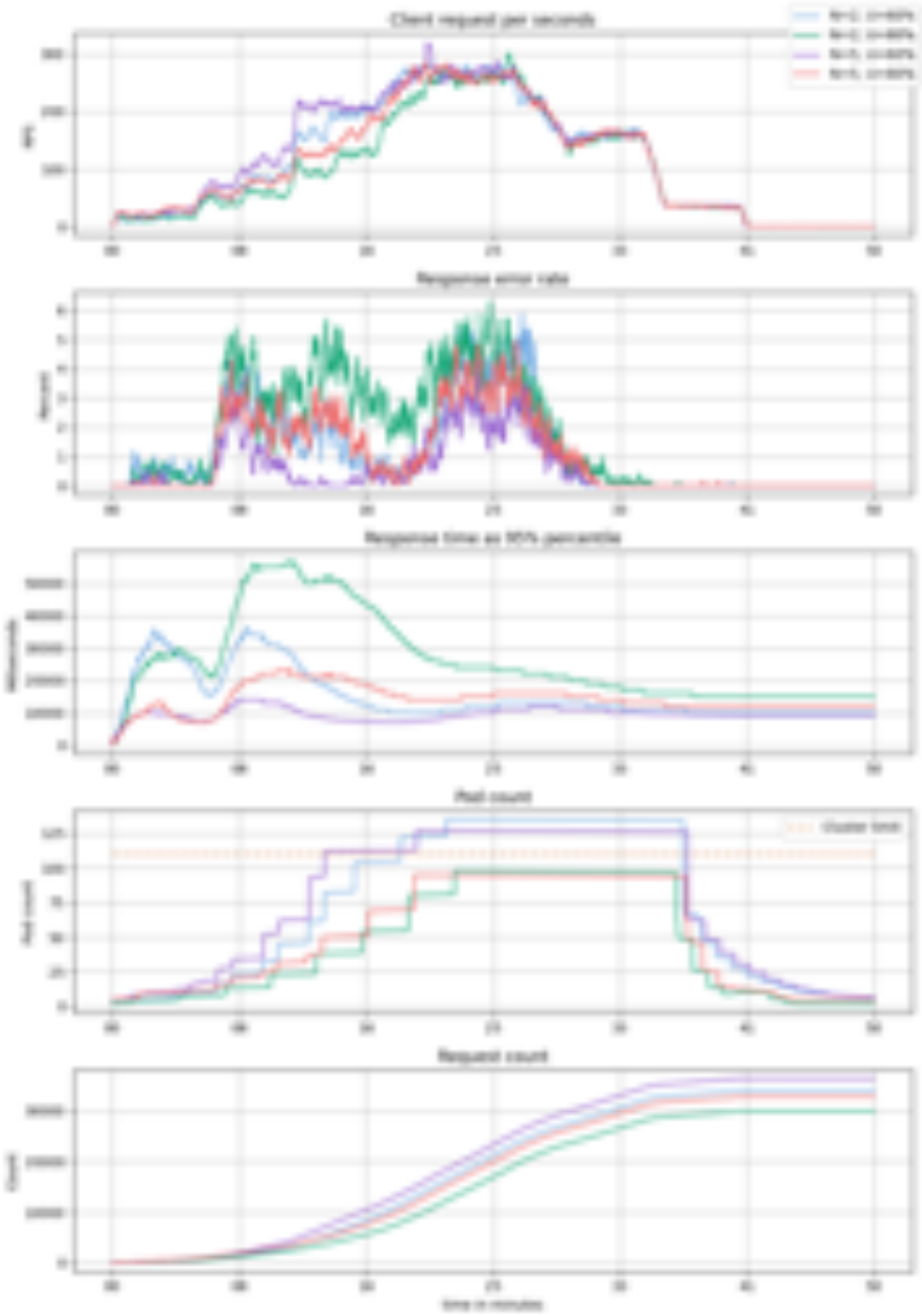


Figure 6.4: HPA statistics

6.4 Resilience concept

To test the Resilience, a chaos engineering approach is taken. Since there is no deterministic chaos, there is neither a happy path nor a synthetic setup. Thus the ODS is used is compared in the integration and production environments (Basiri et al., 2016). This is a direct comparison of the same environment, once with and once without the Resilience concept implemented.

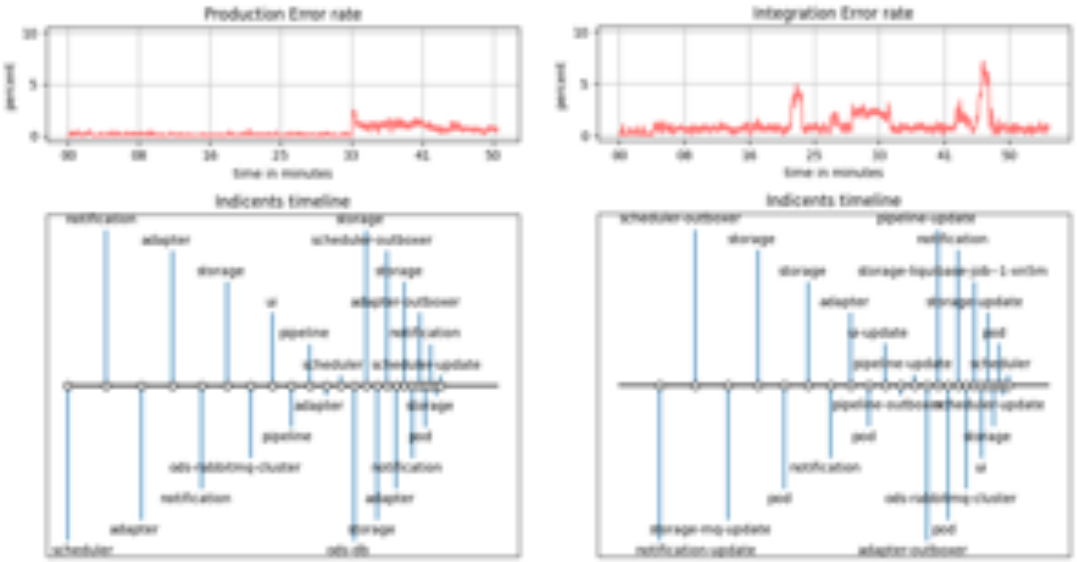


Figure 6.5: Environment stability with REX

A derivative of the chaos monkey, the Resilience EXperiencer (REX), creates the chaos. A Resilience EXperiencer Incident (REXI) is either a random termination of a Pod or a simulated crash of a node caused by stopping the k3s service. The entire REX test is performed for one hour and the REXI frequency is increased linearly from one per minute up to one per every 30 seconds.
Locust is used again for constant load simulation; 20 users with different CRUD operations on the individual ODS services are spawned for this test.

The results of the REX test are shown in figure 6.5. The integration environment has a maximum Error rate of 7.1%, while the production environment has only 2.5%. Production only increased its error rate after one of the duplicated Postgres instances was terminated. In spite of the termination of a RabbitMQ node did not affect the error rate as it did in the integration environment. Besides the more stable runtime, the production ODS was able to serve 25672 requests, while the integration environment only served 24508 (-6.29%).
During the test, there was a bug in the delete endpoint of the Adapter service discovered. The bug was easy to trace via Istio and immediately reported to the developer team.

7 Evaluation

To evaluate the objectives of chapter 3, we match the User Story DoD with the result of the demonstration chapter.

7.1 External Acessible cluster

As demonstrated, the cluster is accessible under a URL and the product owner approves the epic together with all primary user stories:

- **EC-1.1 List of orchestration solutions**
  Kubernetes is the leading implementation that displaced all other orchestrators, hence there is no list but a single orchestrator that 'fully fulfills' all requirements.
  See chapter 5.1.
- **EC-1.2 Multi repo support**
  By using Helm, Kubernetes can handle multiple repositories as container registry.
  See chapter 5.4.3 and appendix A.
- **EC-1.3 Divided infrastructure and application code**
  With ArgoCD, external repositories can provide a single point-of-truth for the cluster state. Repository setups are discussed in chapter 4.3.
- **EC-1.4 Template engine**
  With Helm, the cloc count is reduced from 3618 to 2076 (-57.4%).
  See appendix F.
- **EC-1.5 Testing stage**
  With ArgoCD and namespace configuration, it is possible to instantiate the same application independently in a production and a integration stage.
  Demonstrated in chapter 6.1.

- **EC-1.6 Zero downtime deployment**
  ArgoCD does automatic rolling releases, with zero downtime.
  Demonstrated in chapter 6.2.
- **EC-1.7 Application monitoring**
  Istio gives forms a service mesh and allows detailed traffic tracing.
  Demonstrated in chapter 6.1.
- **EC-1.8 Cluster monitoring**
  Prometheus, Grafana and Longhorn provide sufficient tools for monitoring and alerting the cluster.
  See appendix G and appendix H.
- **EC-1.9 Add server to the cluster**
  The included scripts for k3s provide an easy way to add a agent or primary server.
  See chapter 5.3.2 and the `infra/add_[primary|agent]_node.sh` script.
- **EC-1.10 Remove a server from the cluster**
  The included scripts for k3s provide a simple way to remove a agent or primary from the cluster.
  See chapter 5.3.2 and the `infra/remove_[primary|agent]_node.sh` script.
- **EC-1.11 [Optional] Hybrid cluster setup**
  The k3s add/remove scripts work on Debian based machines independent of the cloud providers. Furthermore, there is a convenience script for AWS instances.
  See chapter 5.3.2 and the `infra/add_agent_node.sh` script.

7.2 Elasticity Concept

As demonstrated, the orchestrator successfully implements a Elasticity model and the product owner approves the epic together with all user stories:

- **EC-2.1 Replication concept**
  Chapter 6.3 gives a detailed analysis of the replication behaviour of the system.
- **EC-2.2 Sharding concept**
  No general sharding concept is implemented, as sharding requires real-world knowledge about application usage. Therefore, priority was given to the provision of a Resilient and thus Elastic database.
  See chapter 5.5.3.

- **EC-2.3 Resource allocation concept**
  There is no vertical autoscaling concept considering all available resources are already used collectively for horizontal scaling. For all of that, allocation of cloud resources is supported.
  See chapter 2.3 and chapter 5.3.3.
- **EC-2.4 Scaling up concept**
  The elaborated concept of HPA replication and Resilience concept is sufficient to make the system Elastic when scaling up.
  See chapter 6.3.
- **EC-2.5 Scaling down concept**
  The elaborated concept of HPA replication and Resilience concept is sufficient to make the system Elastic when scaling down.
  See chapter 6.3.
- **EC-2.6 Failsafe services**
  The HPA replication and Resilience concept is sufficient to make the system Resilient even when a major error occurs.
  See chapter 6.4.

8 Conclusion

To realise the concept of software Elasticity, an enormous amount of technical work had to be done in advance: The transition from monoliths to microservices, the invention of containers, distributed middleware and complex network architectures to connect them all. Therefore, it is not surprising that the implementation of this concept has the same pitfalls, cross-cutting concerns and edge cases as software engineering itself.

This is especially true for bootstrapping a Kubernetes cluster on premise and due to the commercial open source paradigm, some installation issues may even be intentional. However, using Kubernetes as a container orchestrator is practically unavoidable considering that the CNCF has brought together all the major internet companies to compete against AWS - making Kubernetes the most promising orchestrator.

Kubernetes aims for a high level of abstraction, but like all abstractions, they have to trade-off with performance. This ties the implemented Elasticity concept to microservices and Kubernetes. The resulting Kubernetes Rlastic concept is generic and any microservice-based application can dynamically scaled up and down.
The Resilience concept is generic, but the implementation is concrete, as communication between services requires a specific set of middlewares and databases. ArgoCD, Istio and Prometheus with Grafana on the other hand, have a positivy impact on the Elasticity by directly supporting the developer and follow a very generic implemention pattern.

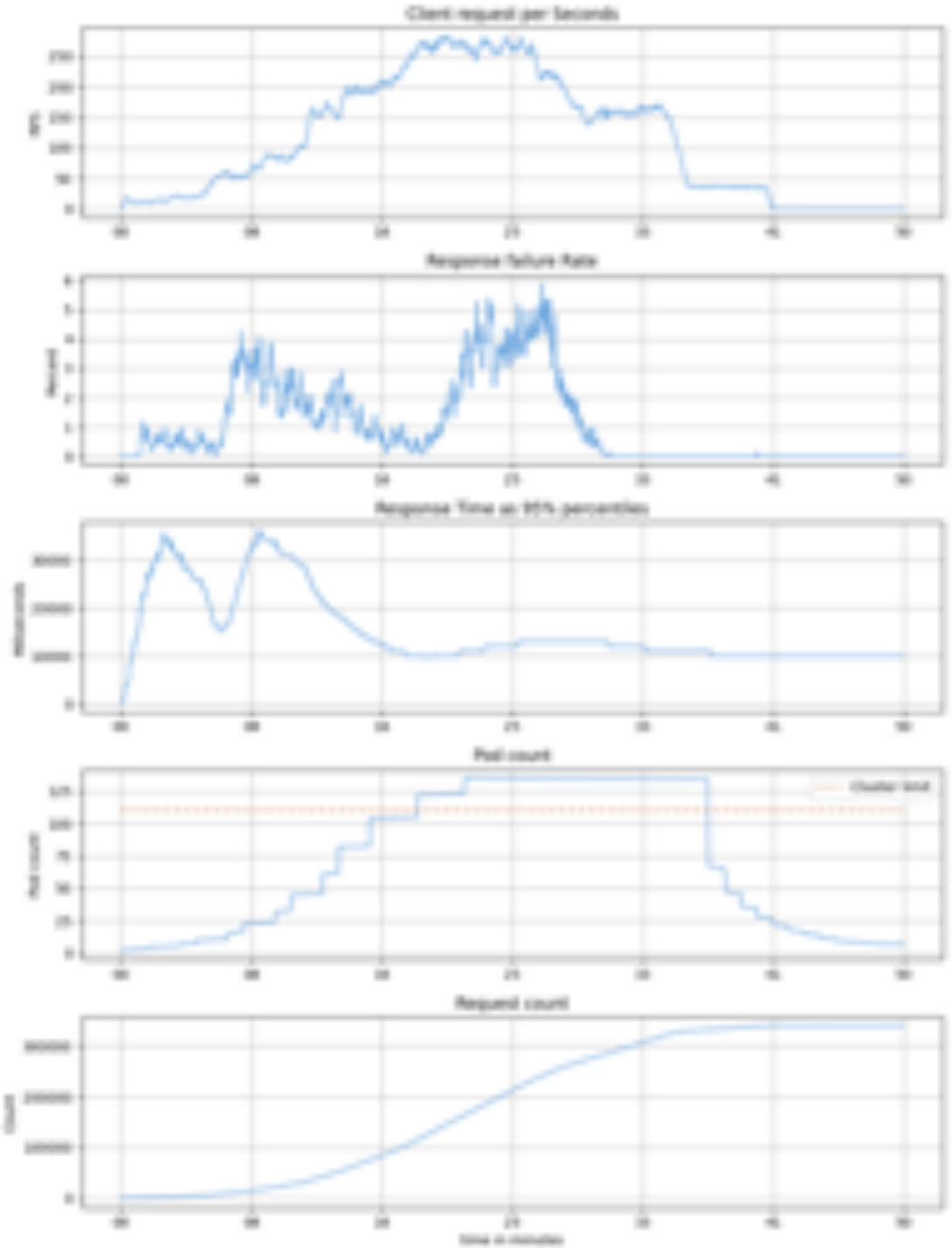
An Elasticity concept by replication might change in the future on account of the microservices design evolves and the underlying Resilience concept with High Availability accomplished with Replica/Primary pattern, changes due to more mature software. Nevertheless, helping developers with real-world application metrics will likely never change the way Elastic software is crafted.

Appendices

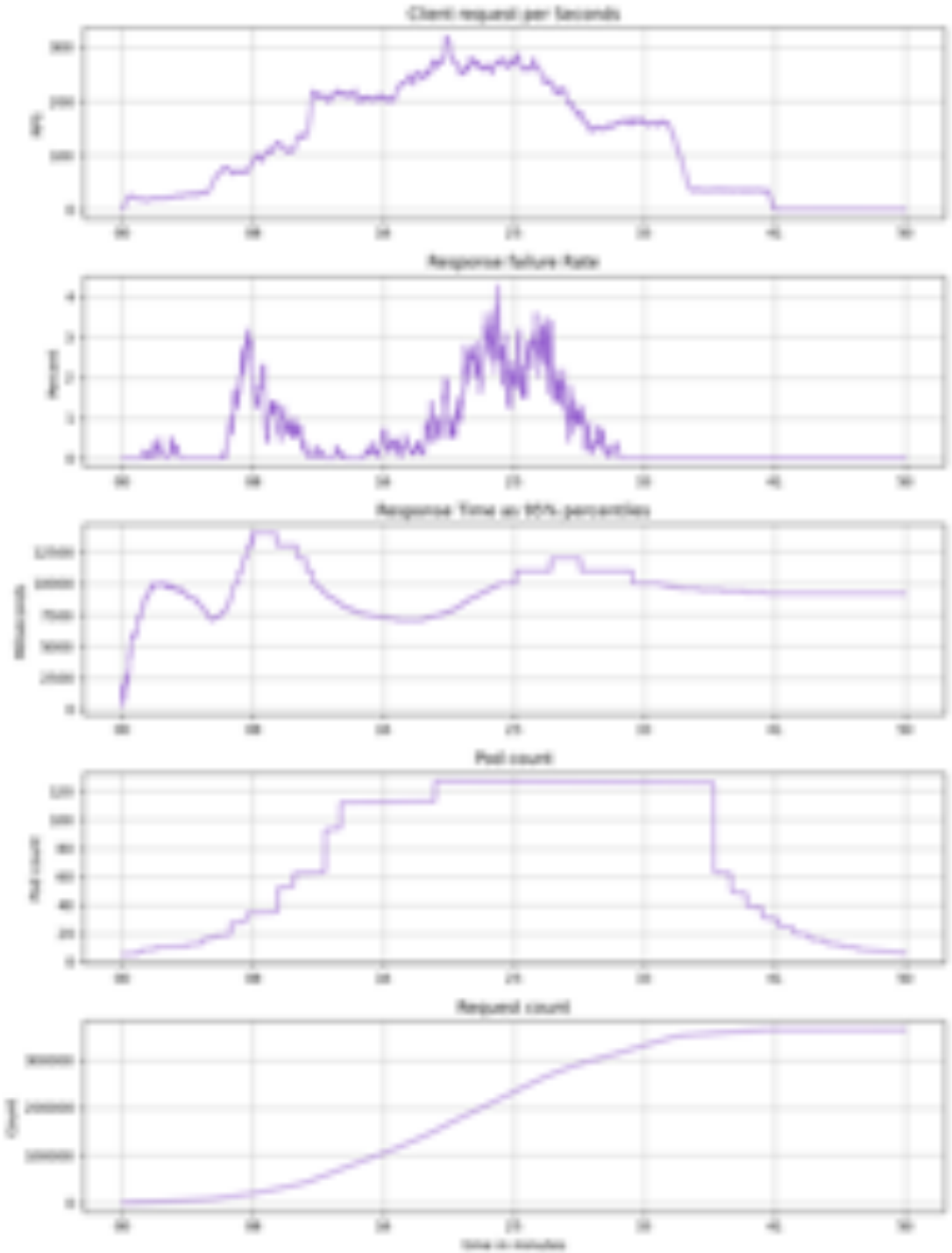
A YAML for Service and Deployment

```yaml
apiVersion: v1
kind: Service
metadata:
  name: notification-service
spec:
  selector:
    tier: notification
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: notification-service
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: notification
  template:
    spec:
      containers:
        - name: notification
          image: "jvalue/open-data-service/notification:latest"
          imagePullPolicy: Always
          resources:
            requests:
              cpu: "100m"
              memory: "64Mi"
            limits:
              cpu: "200m"
              memory: "500Mi"
          env: # SKIPPED
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /
              port: 8080
              scheme: HTTP
```

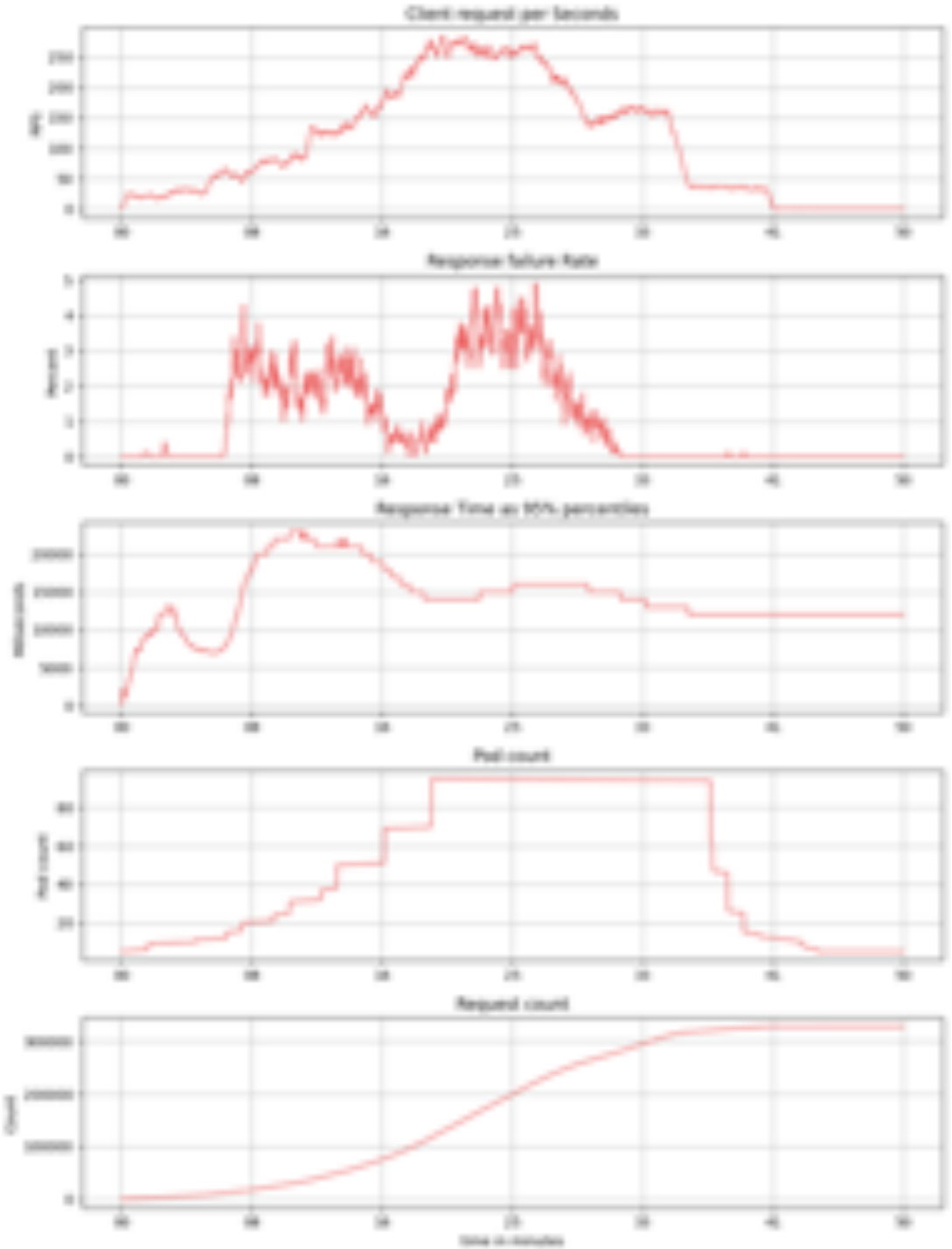
B HPA with N\2, U\60



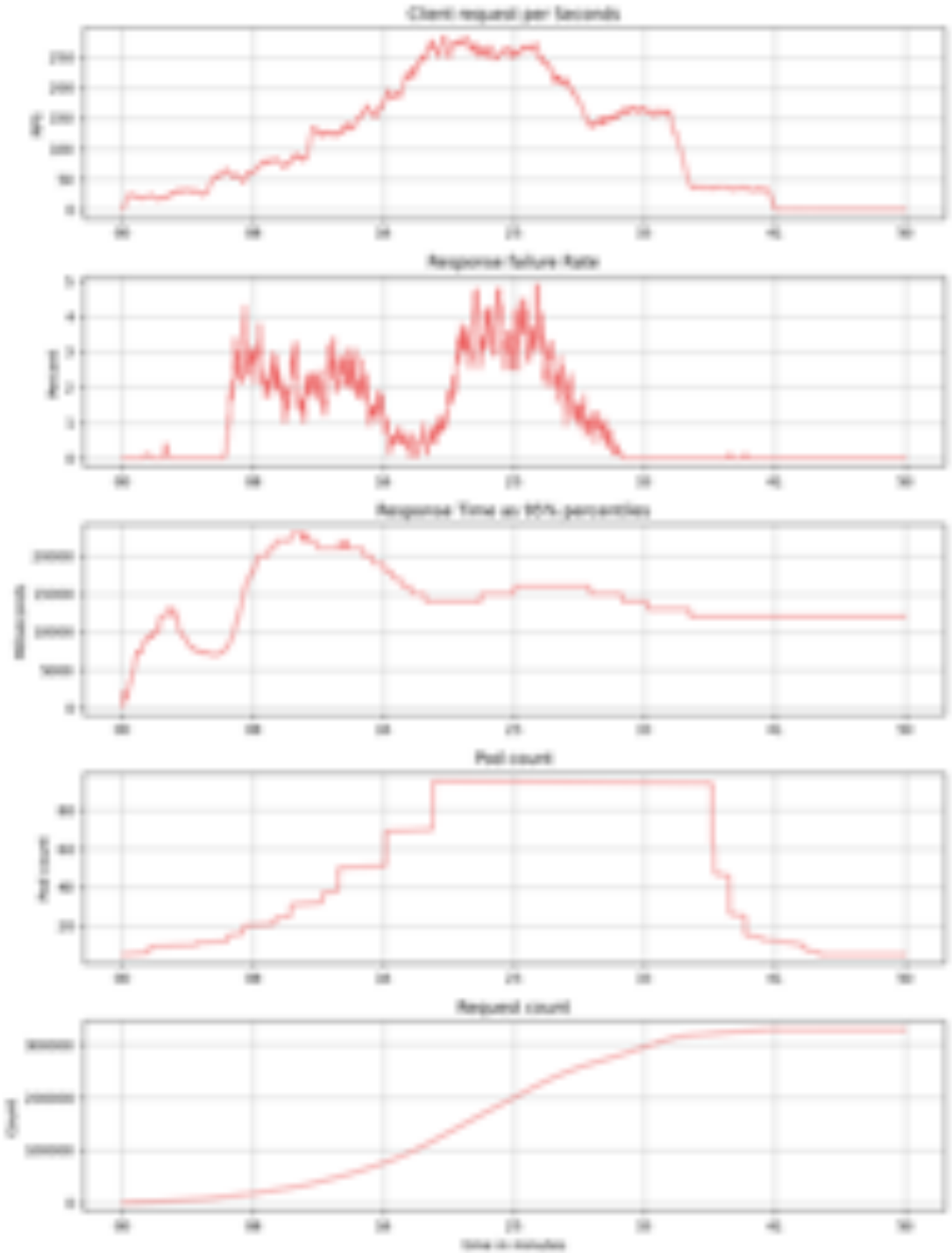
C HPA with N\2, U\80



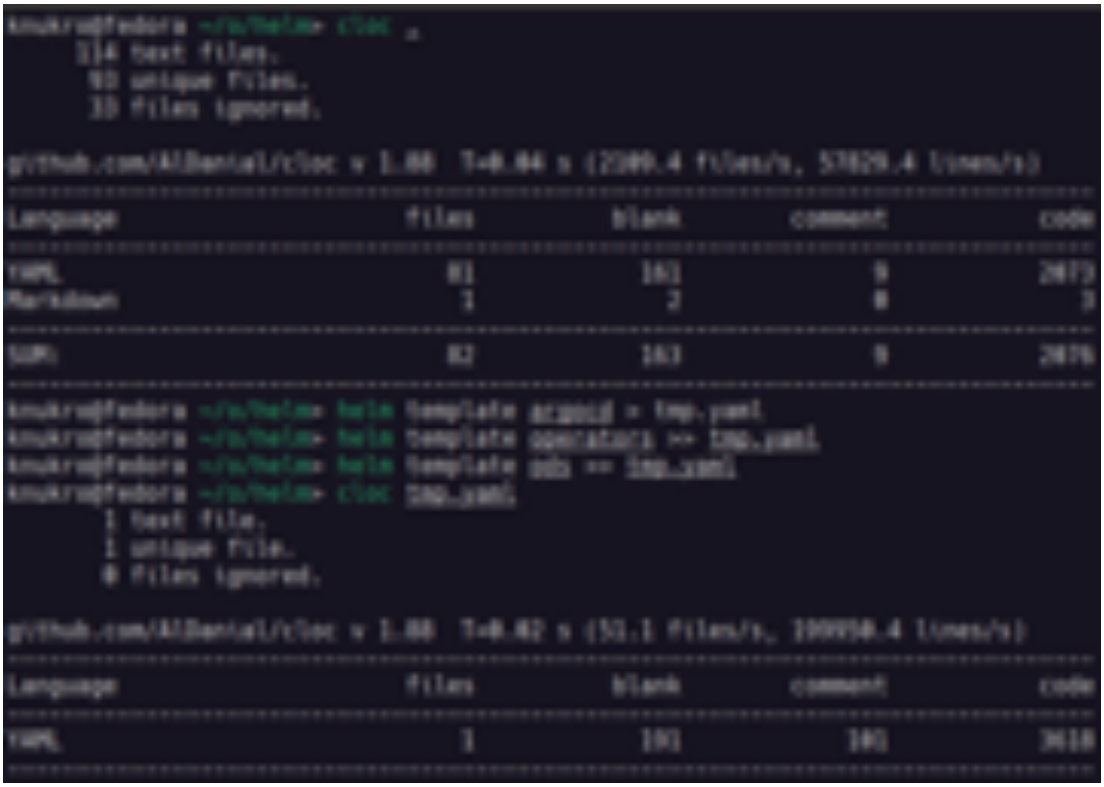
D HPA with N\5, U\60



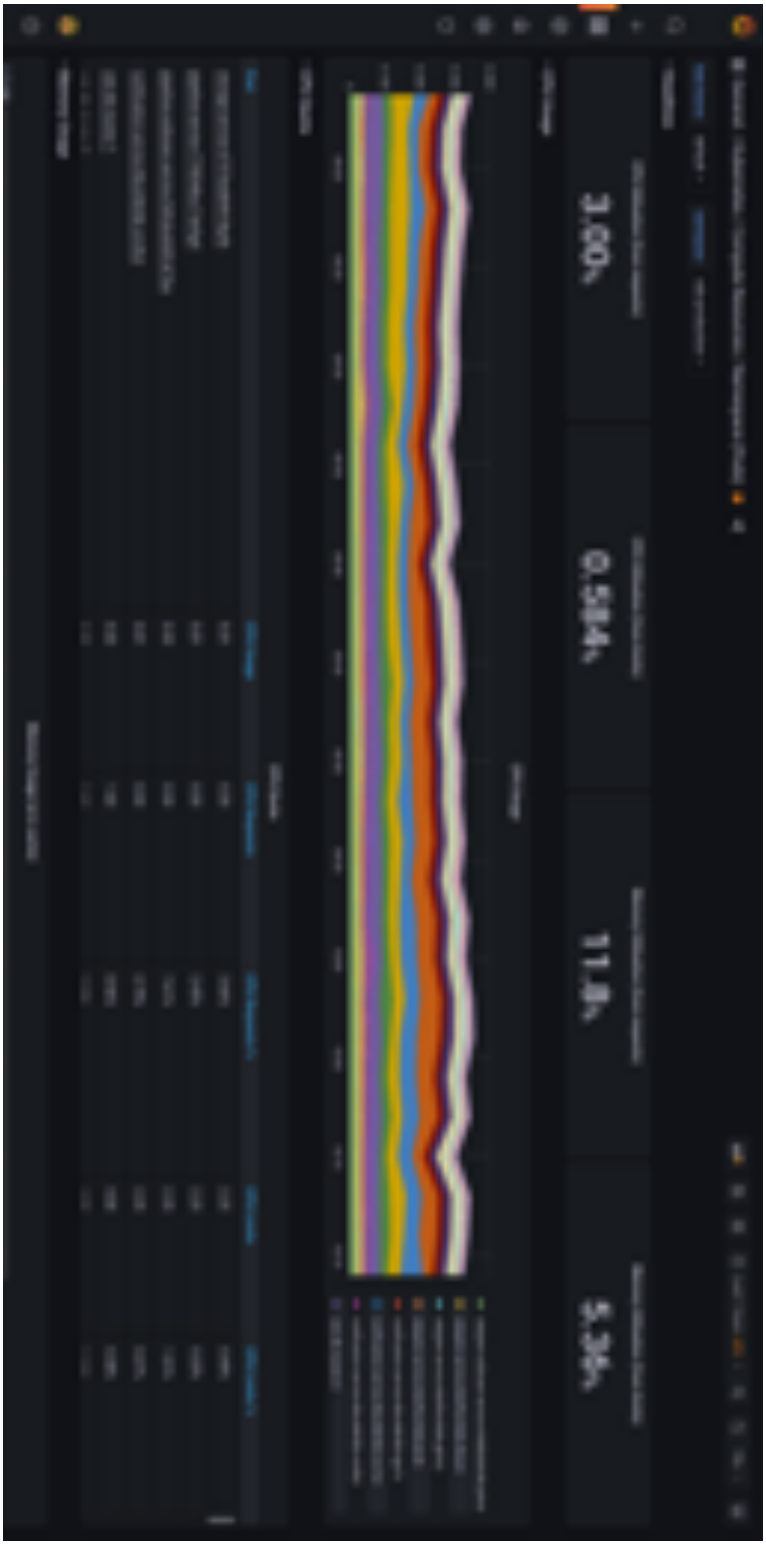
E HPA with N\5, U\80



F Cloc with Helm



G Grafana



H Longhorn

References

Ai, W., Li, K., Lan, S., Zhang, F., Mei, J., Li, K. & Buyya, R. (2016). On elasticity measurement in cloud computing. *Scientific Programming*, *2016*, 7519507. https://doi.org/10.1155/2016/7519507

Arachchi, S. & Perera, I. (2018). Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 moratuwa engineering research conference (mercon)*. https://doi.org/10.1109/MERCon.2018.8421965

Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J. & Rosenthal, C. (2016). Chaos engineering. *IEEE Software*, *33*(3), 35–41. https://doi.org/10.1109/MS.2016.60

Bernaille, L. (2019, August 29). *Kubernetes the very hard way* [Accessed: 2022-14-01]. https://www.usenix.org/sites/default/files/conference/protected-files/lisa19_slides_bernaille.pdf

Chemitiganti, V. (2019). *Kubernetes contributors* [Accessed: 2022-01-02]. https://platform9.com/blog/kubernetes-enterprise-chapter-2-kubernetes-architecture-concepts

Ducastel, A. (2020). *Benchmark results of kubernetes network plugins* [Accessed: 2022-16-01]. https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-august-2020-6e1b757b9e49

Evans, Y. (2017). *Mastering chaos - a netflix guide to microservices*. https://www.youtube.com/watch?v=CZ3wIuvmHeM

Gitops principles v0.1.0. (n.d.). https://github.com/open-gitops/documents/blob/v0.1.0/PRINCIPLES.md

Guide to gitops. (n.d.). https://www.weave.works/technologies/gitops/

Haag, S. & Eckhardt, A. (2017). Shadow it. *Business & Information Systems Engineering*, *59*(6), 469–473. https://doi.org/10.1007/s12599-017-0497-x

Helm documentation [Accessed: 2021-12-20]. (n.d.). https://helm.sh/docs/

Helm homepage [Accessed: 2021-12-20]. (n.d.). https://helm.sh

Herbst, N. R., Kounev, S. & Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not. In *10th international conference on autonomic computing (icac 13)*. https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst

Hightower, K., Burns, B. & Beda, J. (2017). *Kubernetes: Up and running dive into the future of infrastructure* (1st). O'Reilly Media, Inc.

Hilbrich, M. (2019). In microservices we trust — do microservices solve resilience challenges? In *Tagungsband des fb-sys herbsttreffens 2019*. https://doi.org/10.18420/fbsys2019-02

Johannes Schnatterer, D. H. (2021, April 21). *Ciops vs. gitops mit jenkins*. https://cloudogu.com/de/blog/ciops-vs-gitops_de

Kocot, D. & Effing, D. (2021). *Api gateway und service mesh im kontext von service-konnektivität* [Accessed: 2022-01-08]. https://blog.codecentric.de/2021/02/api-gateway-service-mesh-service-konnektivitaet/

Kubernetes documentation [Accessed: 2022-01-02]. (2022). https://kubernetes.io/docs/

Laprie, J. (2008). From dependability to resilience. In *Dsn 2008*.

Martin, R. C. & Coplien, J. O. (2009). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall. https://www.amazon.de/gp/product/0132350882/ref=oh_details_o00_s00_i00

Nadareishvili, I., Mitra, R., McLarty, M. & Amundsen, M. (2016). *Microservice architecture: Aligning principles, practices, and culture* (1st). O'Reilly Media, Inc.

Peffers, K., Tuunanen, T., Rothenberger, M. A. & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), https://doi.org/10.2753/MIS0742-1222240302, 45–77. https://doi.org/10.2753/MIS0742-1222240302

Redhat. (2019, January 8). *What is blue green deployment?* [Accessed: 2022-01-27]. https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment

Riehle, D. (2011). Controlling and steering open source projects. *Computer*, 93–96.

Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E. & Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In *Proceedings of the european conference on computer systems (eurosys)*.

Welsh, T. & Benkhelifa, E. (2020). On resilience in cloud computing: A survey of techniques across the cloud domain. *ACM Comput. Surv.*, *53*(3). https://doi.org/10.1145/3388922

Zhu, J., Li, X., Ruiz, R. & Xu, X. (2018). Scheduling stochastic multi-stage jobs to elastic hybrid cloud resources. *IEEE Transactions on Parallel and Distributed Systems*, *29*(6), 1401–1415. https://doi.org/10.1109/TPDS.2018.2793254