



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

OPERATOR FOR MARIADB GALERA CLUSTER IN OPENSIFT

OPERÁTOR PRE MARIADB GALERA CLUSTER V PROSTREDÍ OPENSIFT

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR

AUTOR PRÁCE

SUPERVISOR

VEDOUCÍ PRÁCE

LUKÁŠ JAVORSKÝ

Ing. FILIP JANUŠ

BRNO 2022

Bachelor's Thesis Specification



Student: **Javorský Lukáš**
Programme: Information Technology
Title: **Operator for MariaDB Galera Cluster in OpenShift**
Category: Databases

Assignment:

1. Get familiar with Linux containers, MariaDB, Galera cluster and OpenShift Platform as a Service(PaaS);
2. Explore and compare existing solutions that offer high-availability for MariaDB database;
3. Design and implement one or more container images based on Fedora packages;
4. Implement an operator managing cluster setup and fail-over autonomously;
5. Prepare stress-tests with simulated crashes, and demonstrate automatic fail-over in the OpenShift environment.

Recommended literature:

- <https://mariadb.com/kb/en/galera-cluster/>
- <https://docs.podman.io/en/latest/Commands.html>
- https://docs.openshift.com/container-platform/4.7/operators/operator_sdk/osdk-about.html

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Januř Filip, Ing.**
Consultant: Horák Jan, Ing., RedHatCZ
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 11, 2022
Approval date: November 3, 2021

Abstract

The main goal of this thesis is to create an orchestrated system, that will adapt and react to the current database server load. The newest OpenShift 4 orchestration and container deployment platform, which is built on top of the Kubernetes API, is being used as a solution to this problem. Using the operator as the mind of the OpenShift cluster, responsible for the deployment, load balancing, detecting crashes, and fail-over recovery in the MariaDB containers, we can ensure that the database will keep functioning, even in the highest peaks throughout the day. Galera replication software built into each MariaDB server ensures that the content of every database in the Galera cluster is consistent.

Abstrakt

Hlavným cieľom bakalárskej práce je vytvoriť riadený systém databázových serverov, ktorý je schopný sa adaptovať a reagovať na ich aktuálne zataženie. Riešenie tohto problému nachádzame práve v najnovšej platforme, určenej pre riadenie a nasadenie OpenShift 4 postavenej na základoch softwaru Kubernetes. Použitím operátora ako mysle celého OpenShift clustra, zodpovedného za nasadenie, vyvažovanie záťaže, detekciu zlyhaní a zotavenie po zlyhaní MariaDB kontajneroch, môžeme zabezpečiť, že bude databáza bezproblémovo fungovať, dokonca aj pri najvyšších špičkách záťaže počas dňa. Na replikáciu medzi databázovými servermi sa využíva softvér Galera vo vnútri každého MariaDB servera, ktorý zabezpečuje konzistentný obsah.

Keywords

MariaDB database, Container, OpenShift, Kubernetes, Galera replication, Automatization, Operator, Ansible, Orchestration, High availability

Klíčové slová

MariaDB databáza, Kontajner, OpenShift, Kubernetes, Galera replikácia, Automatizácia, Operátor, Ansible, Orchestrácia, Vysoká dostupnosť

Reference

JAVORSKÝ, Lukáš. *Operator for MariaDB Galera Cluster in OpenShift*. Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Filip Januš

Rozšírený abstrakt

Cieľom bakalárskej práce je vytvoriť riadený systém databázových serverov, ktorý je schopný sa adaptovať a reagovať na ich aktuálne zaťaženie. Pre vyriešenie tohto problému sa využíva najnovšia platforma, určená pre riadenie a nasadenie OpenShift 4 postavenej na základoch softwaru Kubernetes. OpenShift pridáva klasickému Kubernetes clusteru obohacujúce vylepšenia vo forme jednoduchšieho prístupu a využívania jeho zdrojov. Použitím jednoduchého inteligentného operátora, ktorý operuje nad OpenShift clusterom je možné dosiahnuť tie najlepšie výsledky pri stavových aplikáciách, ktorými sú aj databázy.

V tejto práci sa využíva MariaDB databázový server, ktorý je v minimalistickej verzii zabalený do kontajneru. Docker kontajnery sú známe ako zjednodušené aplikácie, ktoré sú jednoducho nasaditeľné nehladiac od špecifikácií prostredia. Takýmto prostredím môže byť aj využívaný OpenShift vytvorený spoločnosťou Red Hat. Vďaka výborne zosynchronizovaným systémom spoločnosti Red Hat je pre uloženie a následné nasadenie kontajnerových imagov využitý systém Quay.io.

MariaDB databázový server je využitý v práci hlavne pre jeho jednoduchosť a skvelé dopĺňujúce nástroje. Jedným z nich je aj Galera - replikačný software. Galera je software rozšírenie pre databázovú replikáciu, ktorá zaobstaráva, aby každý člen (databázový server) Galera clusteru mal identický kontent. Jej architektúra je založená na Multi-Source hierarchii. V praxi to znamená, že každý člen clusteru je považovaný za hlavného člena a tým pádom nemá žiadny z nich privilegované práva. Obzvlášť užitočná je táto architektúra, pokiaľ je veľká možnosť toho, že by mohol hlavný server nečakane padnúť. V takom prípade nie je potreba riešiť, kto bude jeho následníkom, ale stačí len odobrať odpadnutý databázový server z clusteru.

Navrhnuté riešenie operátora, ktorý sa bude starať o správne nasadenie MariaDB databáz a taktiež vytvorenie a pripojenie do Galera clusteru, je zrealizované pomocou Ansible operátoru. Ansible je jeden z ďalších nástrojov produktového portfólia spoločnosti Red Hat, ktorý je vysoko využívaný v automatizácii nasadzovania, systémového spravovania viacerých zdrojov naraz, alebo sieťového spravovania. Pre správne nastavenie operátora je nutné mať predstavu o jednotlivých krokoch, ktoré bude operátor automatizovať. Táto metóda sa robí pomocou manuálneho nasadzovania kontajnerov do OpenShift clusteru pomocou OpenShift klienta z príkazového riadku. Najdôležitejšia časť, ktorú bude mať operátor na starosti je dohľadanie nad správnym pripojením sa jednotlivých serverov do Galera clusteru. Pre vyriešenie tohto problému bolo treba pridať do kontajnerov systémové premenné, ktoré jednoznačne identifikujú, či sa databázový server pripája do už existujúceho clusteru, alebo má server tento Galera cluster nainicializovať. Ďalej, pokiaľ sa jedná o server, ktorý sa pripája do existujúceho clusteru, musí operátor správne určiť IP adresu serveru, ktorý tento cluster inicializoval. Pre riešenie IP adresy využíva operátor OpenShift services. Tieto services dávajú serverom stabilnú IP adresu, ktorá sa zachováva aj pri prípadnom reštartovaní daného kontajneru v ktorom tento server beží. Operátor teda dohliada na to, aby prvý kontajner, ktorého server inicializuje nový Galera cluster bol nasadený a až po jeho úspešnom nasadení sa môžu začať pripájať ďalšie servery. Počet serverov v clusteri si volí samotný administrátor jednoduchou premennou, ktorú operátor neustále sleduje, a pri zmene jej hodnoty koná na základe požadovaného stavu.

Výsledný operátor a kontajnerové image boli otestované sériou testov. Kontajnerový test má za úlohu postupne pridávať a odoberať členov clusteru a sledovať, či sa tieto zmeny správne aplikujú v Galera clusteri. Tieto testy sú pre ich jednoduchosť testované lokálne, využitím lokálnej Docker siete. Následné testovanie operátora prebieha jeho nasadením, určením hodnoty požadovaného počtu nasadených databázových serverov a ich pripoje-

nia do Galera clusteru. Po manuálnom odsledovaní správnosti nasadenia následuje test náhodného zlyhania jedného a viacerých členov clusteru. Operátor v tomto prípade začne okamžité fail-over zotavovanie vo forme nasadenia jedného alebo viacero nových kontajnerov a ich následného pripojenia do Galera clusteru. Jednotlivé testy odhalili, že operátor a jeho reakcie na neočakávané udalosti fungujú bezproblémovo, keďže po manuálnej kontrole stavu OpenShift a Galera clusterov majú obidva vždy správny počet kontajnerov/členov.

Operator for MariaDB Galera Cluster in OpenShift

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Filip Januš. The supplementary information was provided by Ing. Honza Horák. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Lukáš Javorský
May 10, 2022

Acknowledgements

I want to express my great gratitude to my consultant Ing. Honza Horák, who was guiding me during the whole process, helping me understand the fundamentals of this thesis and always being there when I needed him. Also I want to thank to my supervisor Ing. Filip Januš, who was providing me the information about the formal process of the thesis.

Contents

1	Introduction	2
2	Database	3
2.1	Database management system	3
2.2	Relational database management system	4
3	Cloud native computing	8
3.1	Linux container	8
3.2	Container orchestration	10
3.3	Kubernetes	11
3.4	OpenShift platform	17
4	MariaDB Galera Replication	21
4.1	Replication	21
4.2	MariaDB Galera Cluster	22
4.3	Galera Arbitrator	24
5	Design and Implementation	25
5.1	Existing solutions	27
5.2	Creating container image	28
5.3	OpenShift API workflow	31
5.4	Custom deployment configs	37
5.5	Determining the cluster node IP addresses	40
5.6	Using Kubernetes Service IP address	41
5.7	Automating steps with Operator	42
5.8	Known limitations	46
6	Testing and evaluation	47
6.1	Local container testing	47
6.2	Testing the operator	48
6.3	Test results evaluation	51
7	Conclusion	53
	Bibliography	55

Chapter 1

Introduction

Having a fast and reliable database is one of the top priorities of any organization that needs to store data. As the number of such organizations grows, so does the demand for High Availability systems. The backbone of most systems is their databases, so the requirements for such databases are immense. Chapter 2 goes into further detail about these systems, including MariaDB, which will be used throughout this thesis.

No matter how advanced the database can be, keeping the system up and running during overload becomes increasingly challenging. This is the part when the replication and orchestration of the database servers comes into play. A collection of database servers can then provide the same level of service during the peak and the low loads of the day by eliminating the single point of failure.

The use of the cutting edge technologies in the containerization sphere can help to solve this issue. For example, Red Hat's OpenShift 4 platform does not only provide Kubernetes functionality, but also many additional handy features, which will be discussed in Chapter 3.

The replication function is also an important part of the system that uses multiple database servers. One big data center is not enough if the database needs to be accessed from anywhere in the world and needs to be accessed fast and effortlessly. Multiple data centers are being operated across multiple continents by businesses serving their customers in different parts of the world. Nevertheless, each of these data centers must have the exact same state of data content in their database system. This created the need for engineers to figure out how to accomplish such a task (Database replication). Chapter 4 is dedicated to explaining how the MariaDB multi-master Galera replication works.

Chapter 5 describes the design of the application (OpenShift operator), existing similar solutions and the overall implementation process that leads to the final design. This chapter will also cover every piece of prerequisite work completed on Red Hat's existing MariaDB containers. Apart from that, it also covers possible limitations of the final MariaDB Galera OpenShift operator design.

Chapter 6 and Chapter 7 summarize the results of this thesis and provide great examples of why the MariaDB Galera cluster under the supervision of the OpenShift operator is effective and useful.

Any system administration throughout this thesis will be performed on the *RPM Package Manager (RPM)*-based Linux operating system, like *Fedora* or *Red Hat Enterprise Linux (RHEL)*. System administration in other Linux distributions may differ, so it is highly recommended to use products and tools described in this thesis only on Fedora or RHEL operating system.

Chapter 2

Database

Storing data and being able to redeem them whenever the user wanted was always a challenge. Recording data on paper was unreliable and inefficient, so the engineers were actively trying to improve it as soon as possible. With the arrival of the computing systems, this challenge was getting more and more achievable. Being the first to introduce such a system was a goal for many companies.

And so, in 1960, Charles W. Bachman, designed the integrated database system, the first *Database Management System* (DBMS) [13]. DBMS will be described more in detail in Section 2.1. As was later found [34], these early DBMS attempts were not that impressive, and so the market gave rise to a variety of different types. Nevertheless, one of them is more important, since it has had the biggest impact on modern technology. It was the *Relational Database Management System* (RDBMS), which has its own dedicated Section 2.2.

2.1 Database management system

DBMS is a software package designed to define, manipulate, retrieve and manage data in a database. A DBMS generally manipulates the data itself, the data format, field names, record structure, and file structure. It also defines rules to validate and manipulate this data. [48]

DBMS translates data from its logical representation to its physical representation back and forth[47]. This layer of abstraction brings a much more simple and user-friendly approach to the database systems and makes storing for computers easier. The capabilities of DBMS are depicted in Figure 2.1.

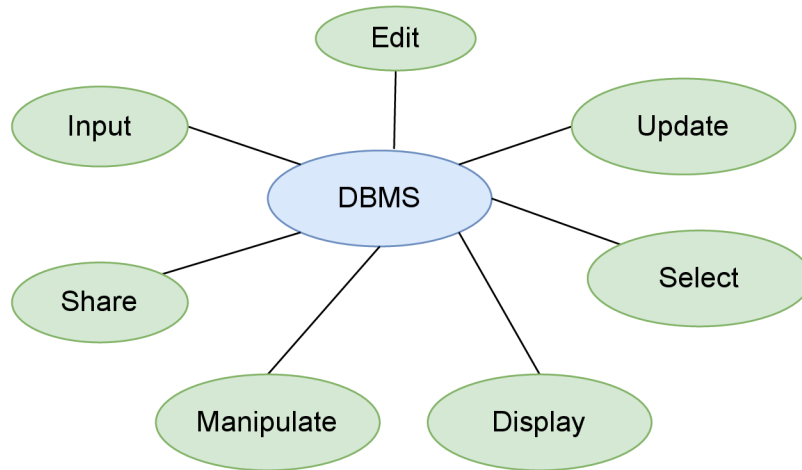


Figure 2.1: Capabilities of database management system simplified.

Users can access and interact with data in the database via DBMS. These actions can range from simply querying data to defining database schemas that affect the database structure. DBMS ensures that users interact with a database securely and concurrently without interfering with each user while maintaining data integrity. [50]

There are many types of DBMS, however, the most common type of DBMS is the Relational one. That is why they will be covered it in a separate Section 2.2.

2.2 Relational database management system

Relational DBMS is a database management system that incorporates the relational-data model, normally including a Structured Query Language (SQL) application programming interface. Both of these are described in Section 2.2.1 and Section 2.2.2 accordingly. The database is organized and accessed according to the relationships between data items. This database is called *Relational database*.

2.2.1 Relational database and Relational model

To understand the RDBMS, it is crucial to understand the Relational Database first. A relational database is a type of database that uses a structure which allows us to identify and access data in relation to another piece of data in the database. Data in the relational database are often organized into tables. [5]

Rows in the database are often called records and columns are called attributes. This model of storing data is called *Relational model*. The database has almost always more rows than columns because the rows are the actual data that are stored.

The relational model has a defined set of constraints, which are mandatory to fulfill. The Domain Constraints, Key Integrity and Referential Integrity. More about these constraints can be found in the article created by GeeksforGeeks [46].

An example of a simple relational database is shown in Table 2.1.

Table 2.1: Example of relational database table. Every table in the relational database is required to have a primary key. A Key is a column (attribute) or group of columns (attributes) used to uniquely identify records in a table. In this case, column “ID” would be the best fit.

ID	Name	Last name	Address	Role
1.	Adam	Sandler	9142 Richardson Street Dracut, MA	Admin
2.	Glenn	Owens	52 N. Grove Street Vicksburg, MS	Admin
3.	Steven	Downy	757 El Dorado Rd. Santa Monica, CA 90403	Developer
4.	Marta	Peterson	82 Old Bald Hill Dr. Williamsport, PA 17701	User

2.2.2 Structured Query Language (SQL)

SQL is a standard programming language specifically designed for storing, retrieving, managing, or manipulating the data inside an RDBMS. SQL has become so popular that it was accepted as an ISO standard in 1987. SQL is the most widely implemented database language and is supported by popular relational database systems like MySQL, MariaDB, and Oracle. [49]. Figure 2.2 depicts how SQL query is handled step by step until it hits the physical database storage.

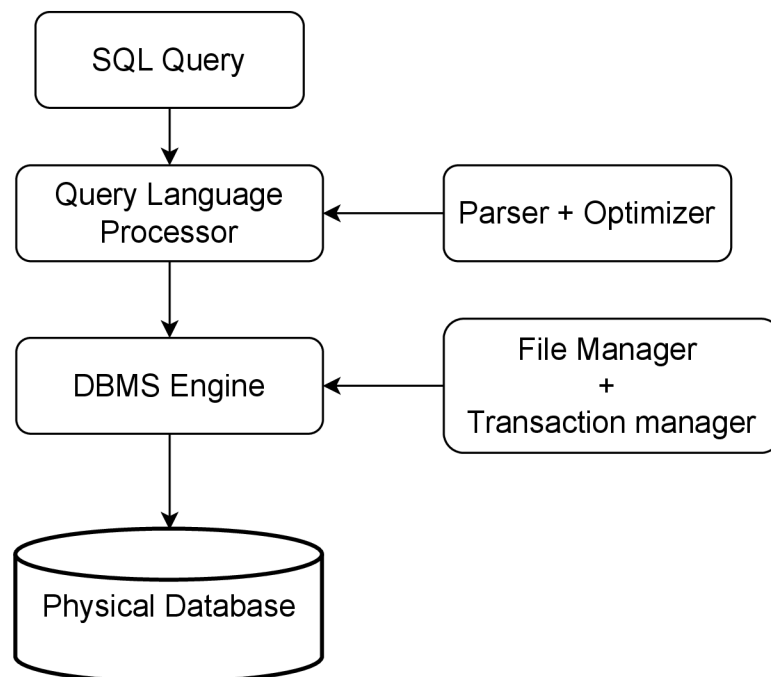


Figure 2.2: A simple diagram showing the SQL Architecture.

SQL statements are used to perform tasks such as update data on a database, manipulate with database, or retrieve data from a database. These statements are divided into a few categories, but only the most important ones will be described.

Data Definition Language (DDL) Statements which perform tasks like Create, Alter, Drop, and ...

Data Manipulation Language (DML) Statements which are used to access and manipulate data in existing schema objects. Statements like Delete, Insert, Select and Update are part of this category.

Oracle describes each of these categories and their statements in greater detail in their documentation [31].

2.2.3 MariaDB and MySQL

MySQL

MySQL is a fully-managed open-source SQL database management system. It is developed, distributed, and supported by Oracle Corporation. MySQL is one of the most used SQL databases in the world. The main features like Scalability, Security, and Connectivity are the most important for MySQL. [29]

The fact that MySQL is supported in every Linux, Microsoft, and MacOS distribution is making it portable and easy to use.

MySQL connectivity between the client and server is forwarded through the 3306 standardized TCP port. This will be later important when the port forwarding method in the OpenShift cluster is discussed in Section 5.3.7.

MariaDB

MariaDB is a community-developed fork from MySQL. It was forked by MySQL developers after they found out that MySQL has been bought by Oracle. It was a fork meant to “ensure that the MySQL code base would be free forever” [33]. These two databases are compatible with each other, although each of them has its own set of features.

One of the features that MariaDB has is the *Galera multi-master replication*. The whole Chapter 4 is dedicated to this feature. Apart from Galera, MariaDB supports more than 10 additional Storage Engines, which are not part of the MySQL server.

Both MariaDB and MySQL software consist of a Server application and a Client application. In the event of data manipulation, the client needs to be connected to the server. The client may only interact with the server after successful authorization.

2.2.4 Server application

The MariaDB server application is used for storing data and serving the clients that connect to it. RDBMS described in Section 2.2 is located on the MariaDB server application. All the database configuration is done on the server.

Configuration file `/etc/my.cnf.d/mariadb-server.cnf` (Distribution Fedora Linux) specifies the options that are passed to the server during startup. Changes in this file will reflect on the server after a restart. In this file, users can specify which storage engine should be used on the database server, for example.

Configuration directory `/etc/my.cnf.d/` is used to store any configuration files for MySQL/MariaDB server or client. To define an application (e.g. server) of the configuration file, the option group needs to be specified first. All of the option groups for MariaDB are described in their documentation [28].

Example of MariaDB Server configuration:

```
$ cat /etc/my.cnf.d/mariadb-server.cnf
[mysqld]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock
log-error=/var/log/mariadb/mariadb.log
pid-file=/run/mariadb/mariadb.pid
```

Some Galera replication options will be added to the MariaDB server configuration file to put the Galera cluster into operation. More about that in Section 4.2.

2.2.5 Client application

A client application is used for interactions with a database server. When getting or storing the data in the database, the user has to do it through the client. The client then forwards any of the user's input to the server.

Most of the time, the word client in relation to MariaDB/MySQL databases refers to a command-line program used to connect to the database server.

In the client/server architecture, the server can handle multiple clients at the same time, however, the client can connect only to a single server at a time. Figure 2.3 shows this type of relation in databases.

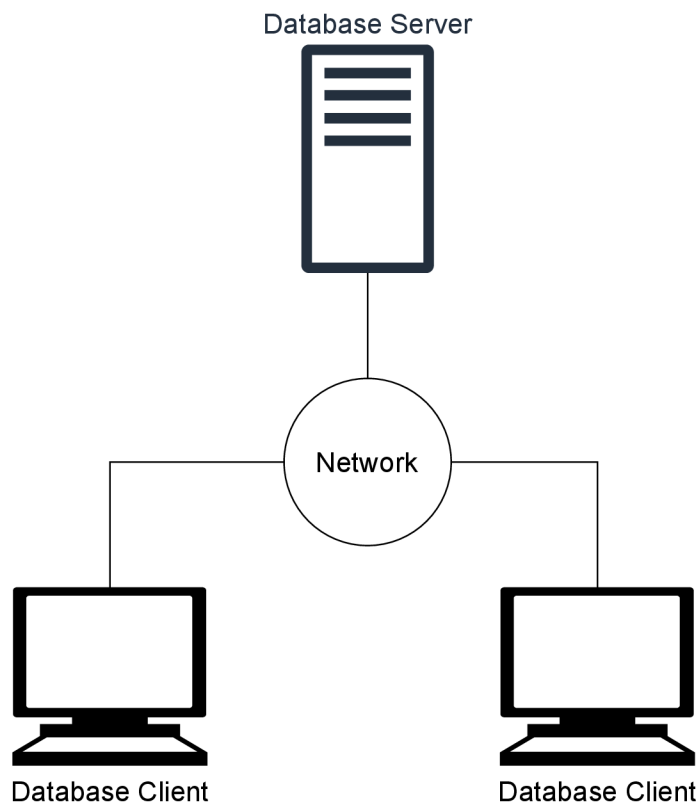


Figure 2.3: Simple example of server/client architecture. This illustration shows clients connected to the server via a network, both local and remote one can be used.

Chapter 3

Cloud native computing

Cloud-native is a modern approach to building and running software applications that take advantage of the flexibility, scalability, and resilience of cloud computing [3]. Using a cloud-native approach has become a huge benefit for many big organizations, and the demand for such systems is increasing every day.

The main goals are to get new services or applications up and running for minimal cost, maximal scalability, and fast deployment. The key to achieving these goals is the automatization of every application's life cycle. Principles like microservice architectures or container orchestration platforms are used for the automation of software life cycle processes [10]. These principles will be discussed in detail in the following sections.

Benefits of a cloud native application [32]

- **Independence:** Cloud native architecture makes it possible to build the applications independently of each other. It also brings the perks of managing and deploying the applications individually.
- **Resiliency:** A well-designed and orchestrated cloud native application can survive events like an infrastructure outage.
- **Automation:** Cloud native applications enable continuous delivery and deployment of software changes that get released on a regular basis. Additionally, developers can use methodologies like blue-green deployment to make changes to the applications without any user experience disruptions.
- **No downtime:** Software update with the right developed orchestrators such as Kubernetes is deployed with essentially zero downtime.

3.1 Linux container

A Linux container is a set of 1 or more processes that are isolated from the rest of the system [43]. All files and requirements are provided from an image, which makes Linux containers portable and consistent. The image is basically just a tar archive containing the main application with all of its dependencies with exception of the Linux Kernel. This also gives them an advantage of quick and easy deployment. Thanks to these benefits, Linux containers have gained big popularity in the IT world.

The environment and configuration in a container are the same, no matter the host operating system. This benefit has been introduced by virtual machines way before. However, containers take advantage of the same benefits without a need for the hypervisor. Containers elegantly use a shared operating system kernel and need somewhat less space in most cases. That is why are containers generally a bit faster and lighter than virtual machines. Nonetheless, virtual machines do have their benefits, for example, the security of a hypervisor against the security of a shared core in containers. Figure 3.1 shows a clear benefit of containers against virtual machines.

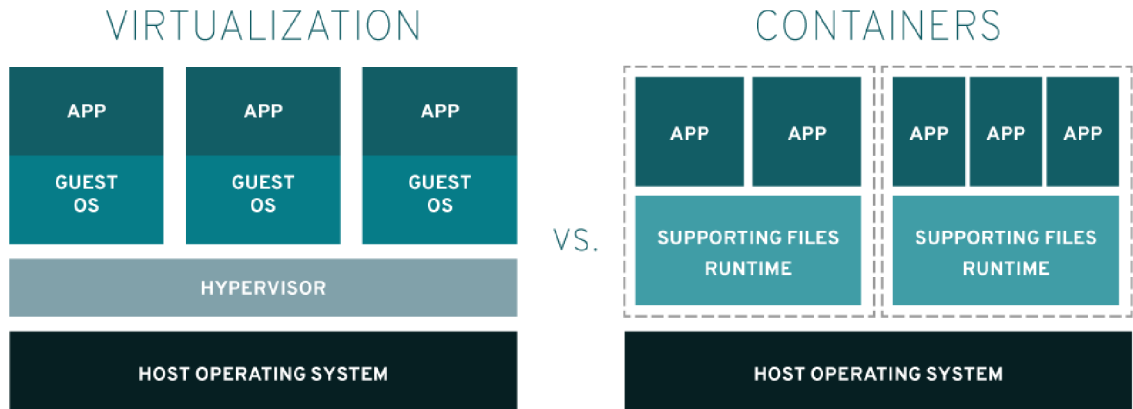


Figure 3.1: Abstract layers of architecture in both virtual machines and containers. Image source [43]

Containers are running instances of a container image. There can be multiple containers running from one container image. Container images are built from the *Dockerfiles* which are described in Section 3.1.1.

Nevertheless, there are certain benefits of virtual machines as well, and the choice between the container and the virtual machines should always depend on the use case.

3.1.1 Dockerfile

As described in the previous section, containers are run from the container images which are built from Dockerfiles. Dockerfiles have a strict structure that needs to be followed in order to successfully build the container image. Tim Butler wrote a great article about Dockerfile and its parts, where he describes it in much more detail [2]. Figure 3.2 serves to visualize what is the relation between Dockerfile, Container Image, and actual Container.

Dockerfile describes what libraries and applications will be installed on the container as well as how the container will be running. There are endless possibilities that can be defined in a Dockerfile, which are described in Docker manual pages [11]. Dockerfile can also describe definitions and environment values (e.g. which TCP ports should be exposed from the service) for applications like Kubernetes or Openshift.

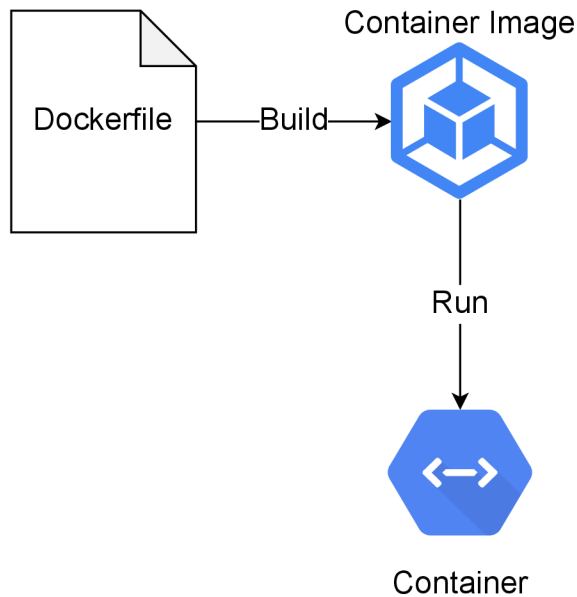


Figure 3.2: Relation between Dockerfile, Container image and Container.

3.2 Container orchestration

Container orchestration automates the deployment, management, scaling, and networking of containers [44]. In the scenario where there is a need for deploying hundreds or thousands of Linux containers, the orchestration can become really useful.

Container orchestration tools provide a framework for managing containers and microservices at scale. There are a lot of container orchestration tools that can be used for managing containers. Popular ones are *Kubernetes*, Docker Swarm¹, or Mesos². Section 3.3 describes the Kubernetes more in detail.

How does container orchestration work in Kubernetes?

The orchestration tool needs a configuration of an application written either in a YAML or JSON file. The configuration management tool then reads the configuration file, which determines where to get the container images, how to establish a network, and where to store logs.

In the event of deploying a new container, the container management tool automatically schedules the deployment to a cluster and finds the right host for it. Management of the container's lifecycle is then the job of the orchestration tool [44].

Applications in real production often use multiple containers, which must be deployed across multiple hosts. Kubernetes is capable of such orchestration, however, it requires a lot of skills and learning to be used. That is why Red Hat OpenShift was created, which will be more discussed in Section 3.4

¹<https://dockerswarm.rocks/>

²<https://mesos.apache.org/>

3.3 Kubernetes

Kubernetes, also known as *K8s*, is a portable, extensible and open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It provides basic mechanisms for deployment, maintenance, and scaling of applications. [25]

A working Kubernetes deployment is called a cluster. A Kubernetes cluster can be visualized as two parts: the control plane and the compute machines, often called nodes. Nodes are running Linux environments that are either part of a physical or virtual machine. Inside each node is running pods, which are made up of containers. This architecture is illustrated in Figure 3.3. More detailed information on components in this architecture is provided throughout the Sections from 3.3.2 to 3.3.6. [45]

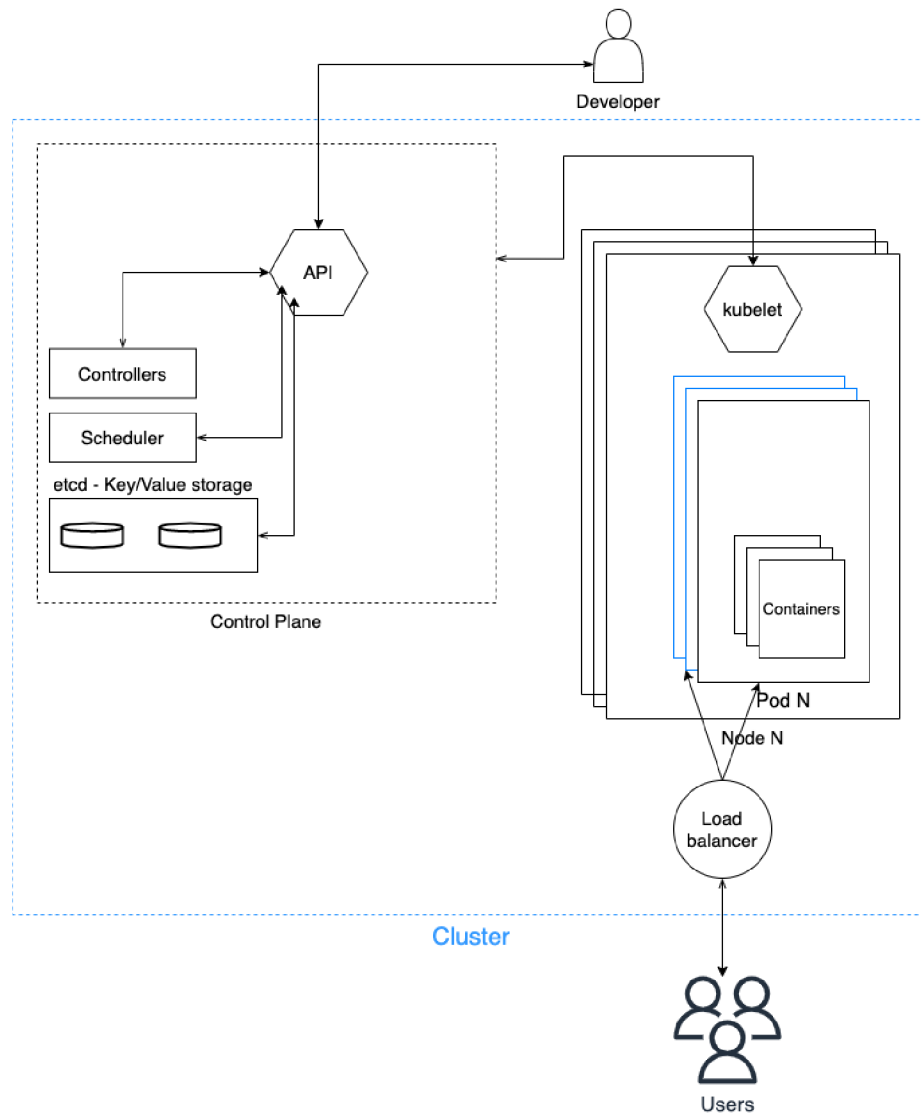


Figure 3.3: Simplified Kubernetes architecture.

Kubernetes also has some features that can be widely used in applications. The most beneficial one is *autoscaling*, described in the following section.

3.3.1 Autoscaling

As demand rises and falls during the day, the load on the applications can vary. In non-autoscaled clusters, this could lead to overloading or underloading. Autoscaling capability means that the administrator does not have to manually provision (or later scale down) the resources as demand changes. It also prevents needless spending. It also ensures that the application will operate with optimal resource utilization and cloud spending. [4]

Kubernetes allows autoscaling at two levels. Pod and Node are two different but fundamentally connected parts of Kubernetes architecture. These two differ mainly in the layer they work with. The most commonly used *Horizontal Pod Autoscaler (HPA)* is displayed in Figure 3.4. HPA allows Kubernetes administrators to dynamically increase or decrease the number of running pods in running applications.

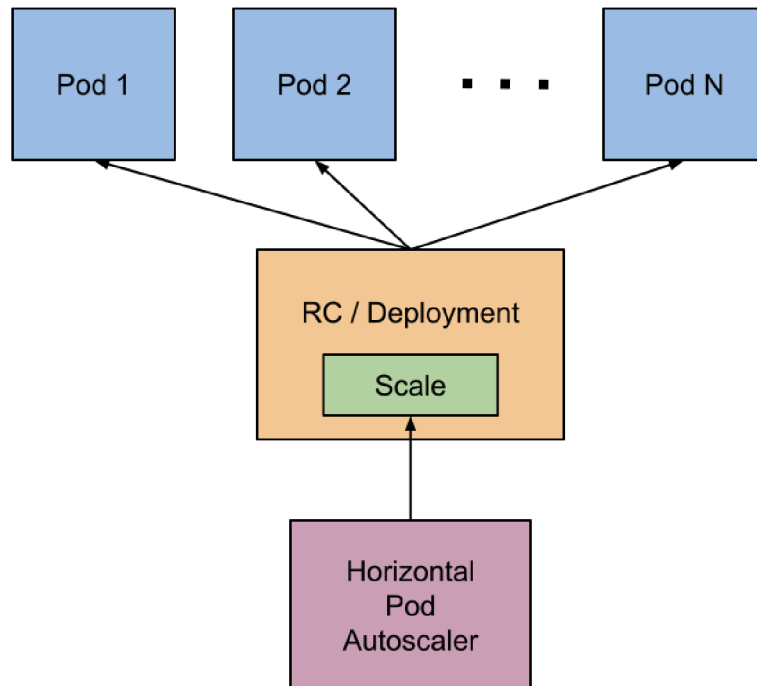


Figure 3.4: Horizontal Pod Autoscaler controls the scale of a Deployment and its ReplicaSet. Deployment and ReplicaSet are described in Section 3.3.2. Image source [21].

3.3.2 Control plane

The Control plane consists of components that make global decisions about the cluster, as well as detect and respond to the cluster's events. Control plane components are typically run on the same machine without any user containers alongside. [22]

API server

If a developer or some application wants to interact with a control plane, it uses *kube-apiserver*, which exposes the Kubernetes API. It serves as the front end for the Kubernetes control plane. The kube-apiserver is designed to scale horizontally if needs to balance traffic.

To interact with Kubernetes API, users can use the *kubectl* command. This command is the main control tool when a cluster admin wants to create, modify, or just check the objects of a cluster.

etcd

The etcd serves as a consistent and highly-available key-value store, for the Kubernetes cluster data backup. Etcd is often run as a multi-node cluster in production, to prevent data loss. [23]

Controllers

Kubernetes cluster has multiple controllers, and each controller is responsible for different events. The most important ones are described below. [1]

- **ReplicaSet:** A ReplicaSet ensures that the right number of Pods are running. If there is a lesser than the desired number of Pods, ReplicaSet creates new Pods to satisfy the desired number. On the other hand, if there are more Pods than desired, they are terminated to maintain the desired number as well. In case of container failure, it will restart its Pod.
- **Deployment:** A Deployment maintains Pods and ReplicaSets with the desired configuration. Whenever a Pod or Replica needs to be shut down (due to memory error or disk corruption), Deployment creates a new one with the same image, command, and configuration. It can be configured for managing updates and rollbacks with minimal downtime. Deployments are managed by the Kubernetes Deployment Controller inside the cluster.
- **StatefulSet:** A StatefulSet keeps the unique identity of a Pod when it needs to be rescheduled. It ensures that the data on a Pod stay persistent even when after a restart.
- **Job:** A Job is responsible for creating short-lived Pods for the purpose of a single task (like Rotating logs) and waits till they finish successfully.

When a controller needs to make any necessary changes in a cluster, it uses the *kube-apiserver* to interact with nodes/pods.

Scheduler

When a new Pod is created, a Scheduler will select and assign a node for it. It also takes things like the resource requirements or hardware/software/policy constraints into account before scheduling. The implementation of a scheduler in Kubernetes is called *kube-scheduler*.

3.3.3 Node

A Kubernetes cluster needs at least one Node to exist, but normally has more. A Node is a machine in Kubernetes, which is either a virtual or a physical machine, depending on the cluster. A Node can have multiple pods within itself, which are managed by a control plane. The Control plane also automatically handles scheduling the pods across the Nodes. Node abstract picture is shown as an illustration in Figure 3.5.

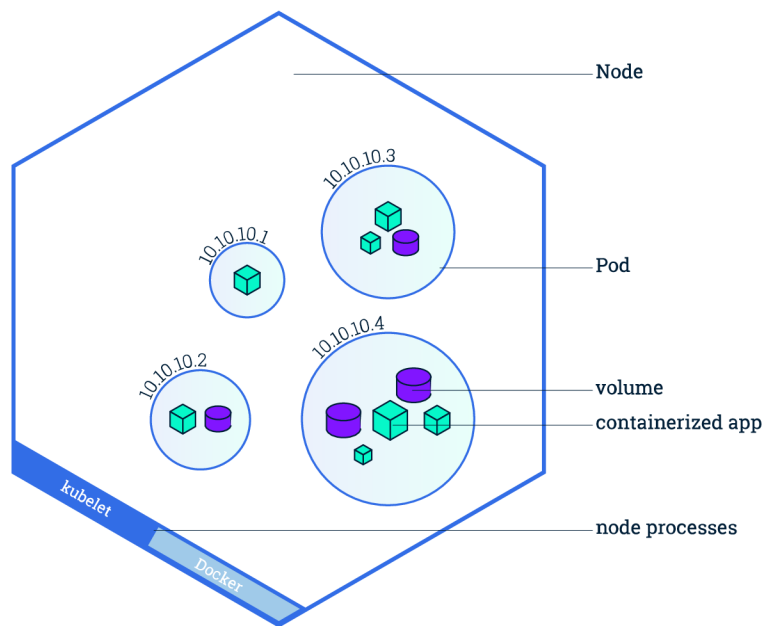


Figure 3.5: A single Node visualization. Image source [24].

kubelet

The kubelet is the primary “agent” that runs on each node. Any changes to the node are forwarded through the kubelet. A kubelet takes a set of PodSpecs that are provided primarily through the apiserver and ensures that the containers described in those PodSpecs are running and healthy. [26]

3.3.4 Pod

When a Deployment (described in Section 3.3.2) is created, Kubernetes creates a **Pod** to host the application instance. A Pod is a Kubernetes abstraction of a group of one or more application containers, with shared resources for those containers. Those resources are *Shared storage (Volume)*, *IP address* and *Information* about how to run each container. [24]

To get a better understanding, Figure 3.6 shows what Pod and its components look like in an abstract way.

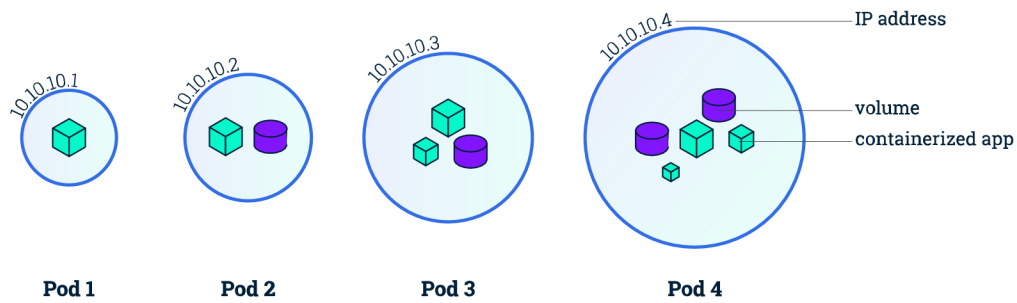


Figure 3.6: A various Pods visualization. It shows that a Pod can consist of a single or multiple containers, with or without volumes. Image source [24].

3.3.5 Service

Pods are instances of various duration (seconds to months) that can be created, destroyed, or can crash due to unexpected errors. However, in production applications, users do not want to always change the IP address used for connection. And so, the Kubernetes has come up with a solution called *Service*. A Service provides a stable and consistent IP address for connections and load balances any requests to the Pods that are listed in this service as its endpoints.

3.3.6 Persistent Volumes

Each Pod does have its own Volume, but it is ephemeral. That means when the Pod is restarted, all data will be lost forever. For some use cases, this does not have to be an issue, but in some like databases, it can cause a huge problems when it happens in production. That is why Kubernetes offers a *Persistent Volumes (PV)*. PV also enables sharing these data across multiple containers in the pod, which can be beneficial in cases like database containers with Apache server containers.

There is also a second type of Persistent volume called *PersistentVolumeClaim (PVC)*. PVC is a request for storage by a user. PVC consumes the PV resources and offers users storage without exposing how the PV is implemented. This abstraction divides a user space from administration space.

As this thesis is mainly covering the proof of concept of the MariaDB Galera cluster in the OpenShift platform, Persistent Volumes are not part of the implementation. However, the database should have a Persistent Volume in case all of the Pods crash or are shut down at the same time. This scenario is more and more unlikely to happen when the number of Pods and Containers is big. Nevertheless, adding the Persistent Volume to the cluster is a great candidate for the future work that will be described in Chapter 7.

3.3.7 Custom resource definition

A resource is an endpoint in the Kubernetes API that stores a collection of API objects of a certain kind [17]

Custom resource definition (CRD) defines a custom object in Kubernetes API. It allows developers to create their own set of objects and define how to interact with them.

CRD can be created and removed from a running Kubernetes cluster through dynamic registration, and cluster admins can modify custom resources independently of the cluster itself. Once a custom resource is installed, users can create and access its objects using the *kubectl* command, just as they do for built-in resources like *Pods*.

3.3.8 Manifest

The manifest is a specification of a Kubernetes API object in JSON or YAML format. A manifest specifies the desired state of an object that Kubernetes will maintain when you apply the manifest. [37]

Kubernetes have two possibilities for how to configure, define and manage its objects (Deployments, Pods, Nodes, ...). It can be done either by manually changing values or using the manifests that are sent to the Kubernetes API server, which is described in Section 3.3.2. The second option is much more common because this option can be easily automated. An Administrator then just needs to write objects description (manifest) in the YAML³ or JSON⁴ format and pass it to the Kubernetes API server. Passing the manifests to the Kubernetes API server is done using the *kubectl* command.

The following listing is an example of nginx **deployment manifest** (in YAML format).

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx
5   labels:
6     app: nginx
7 spec:
8   replicas: 5
9   selector:
10    matchLabels:
11     app: nginx
12 template:
13   metadata:
14     labels:
15       app: nginx
16   spec:
17     containers:
18     - image: nginx
19       name: nginx
20     ports:
21     - containerPort: 8080
```

3.3.9 Minikube

Minikube is a development cross-platform tool for deploying Kubernetes applications. In a matter of minutes, a user can configure the local Kubernetes cluster using Minikube. This

³<https://yaml.org/>

⁴<https://www.json.org/>

is especially beneficial in the early stages of development, so the developer does not have to deal with login and network problems with an actual Kubernetes cluster.

Minikube cluster is deployed in the local Docker/Podman container, however, can be configured to use virtual machine as well. Kubernetes official docs are great for understanding the basics of Minikube and for starting the first minikube cluster [19].

3.3.10 Kind

Similar to the Minikube, *Kind* aka “Kubernetes-in-Docker” is a tool for running local Kubernetes clusters using a Docker container [18]. In contrast to Minikube, Kind uses Docker containers instead of a virtual machine, which results in a faster startup time.

Official Kubernetes operators community consider *Kind* a better local OpenShift/Kubernetes cluster solution because of the strange behavior of the Minikube internal network also called „Minikube weirdness“.

3.4 OpenShift platform

Red Hat OpenShift is an open-source container orchestration platform designed for enterprises based on Kubernetes. It includes a few container technologies, like OpenShift container orchestration software, which is built on top of the OKD (community variant of OpenShift Container Platform) project [35]. OpenShift adds layers of security features and abstract commands that make working with Kubernetes a lot easier. These benefits are crucial for large enterprises and especially useful when used in the hybrid cloud⁵.

OpenShift Container Platform is a private platform-as-a-service (PaaS) platform which enables developers to develop and deploy their applications on a cloud infrastructure. It runs on the Red Hat Enterprise Linux (RHEL) operating system and a set of Docker-based application containers managed under Kubernetes orchestration described in Section 3.3.

OpenShift provides a set of features that are widely used in production. Features like *Management of multiple clusters*, *Integrated CI/CD pipelines* and *Automatic installations and upgrades* helps developers to deliver and maintain the product with the newest updates while not having periodical outages.

However, OpenShift has some limitations compared to Kubernetes. The main limitation, in this case, would be the supported operating systems. Kubernetes supports almost every Linux distribution OpenShift does support only RHEL, CentOS, and a couple more Red Hat alternatives.

oc command

OpenShift command-line interface (CLI), the *oc* command, is used for developing and managing applications in the OpenShift Container Platform. OpenShift offers web console development as well, but it may lack the straightforward approach of the *oc* command. Even though the OpenShift itself is built on top of the RHEL operating system, the OpenShift CLI can be used in any kind of environment thanks to its Golang-based libraries, used to communicate with the OpenShift cluster. This command is also commonly referred to as an OpenShift client.

⁵Hybrid cloud - computing environment combining third-party public cloud and local private cloud. There can be multiple private and public clouds in the architecture.

3.4.1 OpenShift projects

Most of the time, the applications running in the OpenShift cluster need to have some kind of a sandboxing. This prevents other applications from using or editing their Pods, Deployments, Replications, and other resources specific to the project.

To create a new OpenShift project, the *oc* provides subcommand *oc new-project*. Basically, a project needs to have its unique name, and the best practice is to include a summary of the project in its description. There are other options to do with the project, which can be found in the official documentation of OpenShift projects [42].

3.4.2 Image streams

Image streams are used in the OpenShift Platform, which serves as associated tags for container images. Image streams are bringing an excellent feature within themselves. It allows administrators to see what images are available and choose the specific image they need, even if the image in the repository changes.

Image streams do not contain the actual images, they are just a single virtual view of related images, similar to an image repository. Fundamentally, they serve as a record table for container images. For example, if the **mariadb** container image has multiple versions (tags), they all can be stored in the image stream. These tags are then easily accessed from the image stream using an analogy like **mariadb:10.5**, where **10.5** represents a tag, and **mariadb** represents an image stream where this tag is stored.

Deployments and builds can be configured to watch an image streams. When an update is added, it can be updated automatically or skip this update. It depends on the configuration.

3.4.3 Operator

Operators are a method of packaging, deploying, and managing a Kubernetes application.[41].

The Kubernetes application is an application both deployed and managed by Kubernetes API server using the *kubectl* (Section 3.3.2) or *oc* command. Operators are automating the manual work of the OpenShift application administrator. An operator is watching over a Kubernetes environment (OpenShift Container Platform) and using its current state to make decisions in real-time. Operators can also be configured to handle upgrades seamlessly and react to failures automatically with predefined strategies. Basically, the Operator is an advanced controller that is operating with the Kubernetes cluster according to its current state. It is a human developer put into the binary file. The operator runs in a loop, so any changes in a cluster or in a configuration are taken care of immediately.

The stateless applications may be just fine with the Kubernetes automation itself. They can easily work with a minimal number of human operational actions. However, in the world of stateful applications, challenges that need immediate action become more and more common. Databases, for example, need a lot of manual work, which is not ideal for an “automotive” environment like Kubernetes. That is where Operators come in place. Figure 3.7 depicts the Operator and the way how it handles the Kubernetes platform objects (Section 3.3.3) and custom resource (Section 3.3.7).

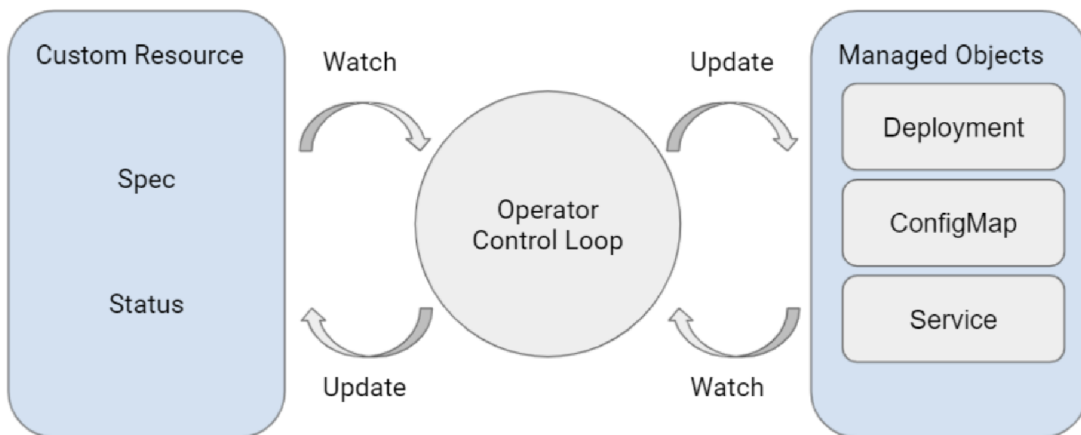


Figure 3.7: Operators (control loop) interactions with different parts of Kubernetes platform control plane, as well as Custom Resource. Image source [51]

3.4.4 Operator Framework

The Operator Framework was released shortly after the Operators. Many believe that the Operator Framework will bring another set of tools to the existing Operators and their workflows.

Operator Framework includes:

- Operator SDK:** Operator SDK makes it easy for developers to build, test, and package Operators without requiring experience or knowledge of the Kubernetes API. It connects application logic with the Kubernetes API, easing development work for developers. Only three kinds of operators are currently supported by the SDK. The first is an operator written in pure Go. With the help of tools like, *kubebuilder* and *kustomize* to scaffold and generate the operators functions. The tools can only generate simple and repetitive parts of the codes, so the logic needs to be written in the Go programming language. The second type is using an open-source management tool called Ansible. The Operator SDK provides an option for creating an Operator by using existing Ansible playbooks and modules to deploy Kubernetes resources without having to write any Go code. Ansible operator could use roles instead of playbooks for easier compatibility between versions, however they are harder to create. Lastly, the Helm-based operator (which unfortunately does not support scaling, auto-config tuning, or failure recovery) is the least feature-rich of the three options. Due to its easy development and widely supported features, an Ansible-based operator will be used throughout this thesis. Section 3.4.5 provides further detail about the Ansible framework.
- Operator Lifecycle Management:** Operator Lifecycle Manager (OLM) provides rich installation and update mechanisms for all of the Operators running across a Kubernetes cluster. It is used for managing and installing the operator into production. [36]

Operator SDK with Ansible

As mentioned above, the *operator-sdk* tools is used in automating most of the code creation for Kubernetes operators. It can scaffold⁶ most of the YAML template code that is later used in production. It can also create a webhooks for the API server to control what is being sent to it, before it is applied.

When user uses the *oc* command, that was described in Section 3.4, it automatically sends the HTTP requests through the REST API to the server. However, these requests has to be created from scratch when creating an operator. In order for the operator to communicate with the API server, it must have properly structured HTTP requests. To ease the work of creating every single request operator-sdk with the help of a Go libraries can just easily call a certain function with proper parameters.

Another great feature of operator-sdk is the automated manifest generation. The developer has to define the key values and parameters for the manifests, and operator-sdk will generate the whole template with these values applied. As operator-sdk is very heavily Makefile driven, it only takes one or two *make* commands, and the generation process is done. Makefile also offers a lot of deployment rules to ease the work for bundling or pushing operators into the repository.

Operator-sdk offers great structured project generation. It can generate anything from the custom resource definitions (more in Section 3.3.7) to the controllers and example manifests for the testing deployment. These files are then automatically distributed to the corresponding folders.

The Ansible operator uses Golang libraries under the hood and is easy to create. As this operator is the alpha version used as a proof-of-work, it is best to use the Ansible operator.

3.4.5 Ansible

Ansible Automation Platform provides an enterprise framework for building and operating IT automation at scale, from hybrid cloud to the edge [40].

Red Hat's Ansible software is widely used in automation. Everything from machine deployments to system administrations to network configurations can be automated using Ansible. Ansible uses so-called *playbooks* to configure the automation purpose. Playbooks are basically just a YAML file with a specific Ansible syntax.

3.4.6 Quay.io

Container images (Section 3.1) can be build anywhere. Unfortunately, one problem stands in the way of this ability. Transferring the container image to any machine. Fortunately, container image registries are excellent at solving this problem. A developer can just simply build the container image and then push it to the registry (kind of like a git push works). There are several options for these registries, like *Docker Hub*, *Azure Container Registry* (ACR), or Red Hat's *Quay.io*.

To ensure the best connection and performance, Quay.io will be used to store all of the container images created for this thesis. There is no need to have these images private, so the public ones will suffice.

⁶code generation technique

Chapter 4

MariaDB Galera Replication

4.1 Replication

Replication is the process of copying data from a central database to one or more databases. [14]

Load balancing in database servers is very important. Using multiple database servers to serve customers is a common scenario. But when something changes in one of the databases, this change needs to be reflected on all of them as well, otherwise it could lead to a very unpleasant situation. That is where *Replication* comes in place. Replication is a process when the main publishing database (**Source** or historically Master) is copying (replicating) the newest data to all of the subscribing databases (**Replicas** or historically Slaves). This type of database replication architecture is called **Source-Replica** (Master-Slave) replication.

Even when the Source-Replica replication is the most commonly used type of replication architecture, it has its disadvantages. For example, when the Source fails, the new Source needs to be chosen from all Replicas (not an easy task), which causes longer downtime of the whole database cluster. The replicas are read-only instances, which could lead to other problems in the production. It definitely has advantages like easy configuration or great isolation of Replicas from Source.

To eliminate some of these disadvantages, the **Multi-Source** (Multi-Master) replication was created. When a client needs to write something to a database, it can be done on any Source node in the cluster. This possibility is very important in times when client write requests need to be load-balanced across the cluster. Also, Multi-Source replication handles the failover scenarios quickly and automatically (no need to vote and later promote, Replica to Source). Of course, there are some disadvantages to this architecture as well. Complicated configuration or loose consistency is on the top of the list. Figure 4.1 shows how these two replication architectures differ.

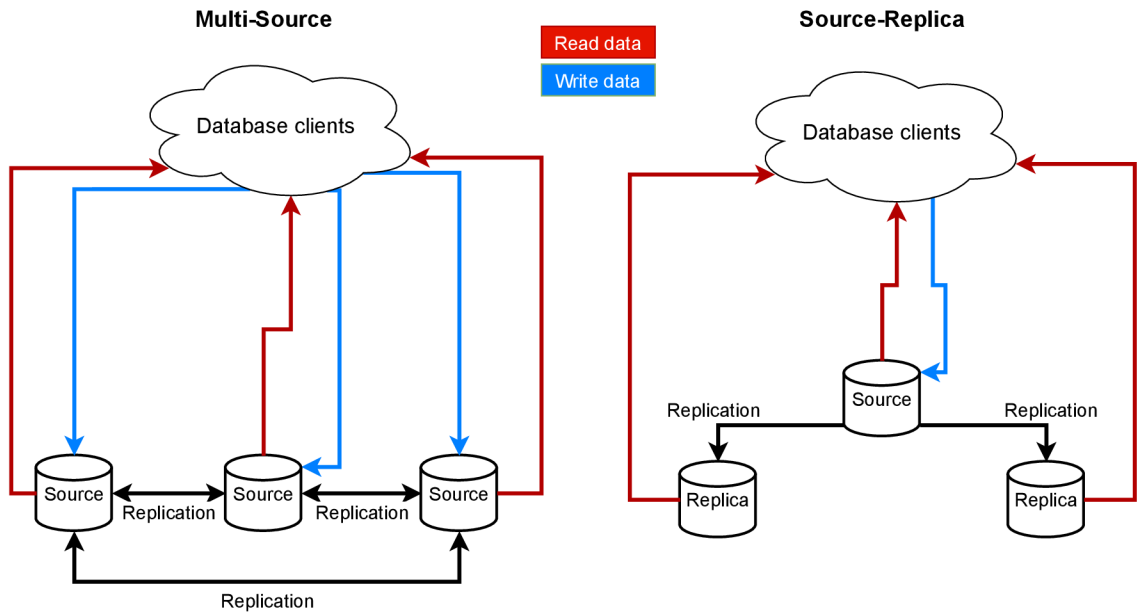


Figure 4.1: Comparison between Multi-Source and Source-Replica replication architecture.

4.2 MariaDB Galera Cluster

MariaDB (more detail in Section 2.2.3) has the Galera replication software built inside of it. It is a multi-source cluster available on Linux-based operating systems only. Galera software is capable of automatic membership control, which also automatically drops the failed nodes from the cluster. It also contains features like Synchronous Replication (guarantees that if changes happened on one of the nodes in the cluster, they happened on other nodes synchronously), Traffic Encryption, and no Source-Replica Failover Operations.

To join the MariaDB Galera cluster, the database server needs to be started with several options (Options in the MariaDB server are described in Section 2.2.3). Options that are mandatory in order for Galera Cluster to be enabled and to work properly with MariaDB are in the listing below.

```
$ cat /etc/my.cnf.d/mariadb-server.cnf
[mysqld]
wsrep_on=1
wsrep_provider=/usr/lib64/galera/libgalera_smm.so
wsrep_cluster_name="my_wsrep_cluster"
wsrep_certify_nonPK=1
binlog_format=ROW
innodb_autoinc_lock_mode=2
default-storage-engine=innodb
wsrep_cluster_address=gcomm://
```


When the first node (database server) initializes the cluster, it has to be started with the `-wsrep-new-cluster` option. All other nodes that want to join this cluster need to modify option `wsrep_cluster_address`, so it contains the IP addresses of nodes in the cluster. For example, like in the listing below. The MariaDB servers with the IP addresses `10.0.7.10` and `10.0.8.11` are already part of the Galera cluster.

```
wsrep_cluster_address=gcomm://10.0.7.10,10.0.8.11
```

Galera Replication library is a shared library that can be linked with any transaction processing system, which implements the wsrep API hooks. For replication, Galera uses these TCP ports.

- **Galera Replication Port** (default: 4567): This port is used for Galera Cluster replication traffic
- **Incremental State Transfers (IST) Port** (default: 4568): In IST transfer, the cluster provisions a node by identifying the missing transactions on the joiner and sends them only, instead of the entire state. [8]
- **State Snapshot Transfer (SST) Port** (default: 4444): In SST transfer, the cluster provisions nodes by transferring a full data copy from one node to another. When a new node joins the cluster, it initiates SST to synchronize its content with the rest of the cluster nodes. [8]

wsrep API

Wsrep API is a replication plugin interface for database servers. In the wsrep API replication model, the database server needs to have a state. State refers to the contents of the database. Whenever clients modify the database content, its state is changed. In a Galera database cluster, all of the nodes have the same state. If not, they synchronize it with each other by replicating and applying the changes in the same serial order [7]. Visualization of the wsrep API is in Figure 4.2.

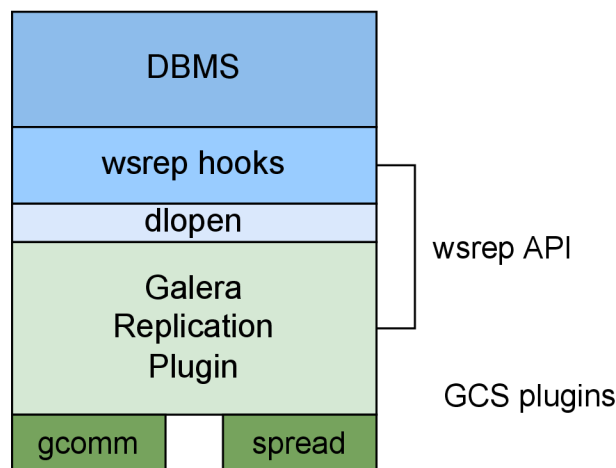


Figure 4.2: Galera Replication wsrep API layers displayed to a layer model.

4.3 Galera Arbitrator

When deploying a Galera cluster, it is recommended to have at least three instances. This could mean having three nodes or even three data centers. However, building a new data center or buying a new machine to serve as a database node could be expensive. To avoid these costs *Galera Arbitrator* can be used instead. Galera Arbitrator is a member of a cluster that participates in voting but not in the actual replication. [6]

Galera Arbitrator serves as a member of the cluster when having an odd number of nodes. It helps to avoid split-brain situations when deciding which node has the most recent content and should be replicated onto other nodes (this decision has to be made when one of the nodes fails). It can also request a consistent application state snapshot, which can be used to make a database backup. Figure 4.3 depicts the use of Galera Arbitrator in a decentralized cluster.

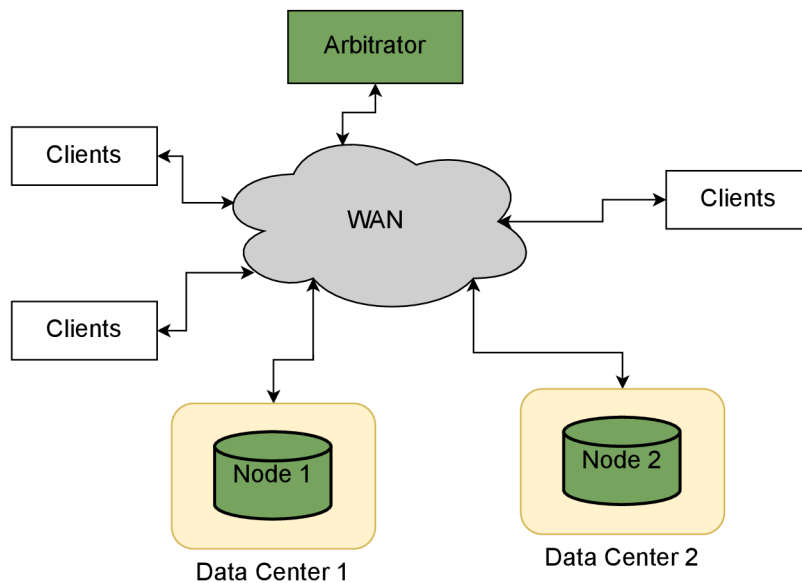


Figure 4.3: Galera Arbitrator used in a non local Galera cluster.

Chapter 5

Design and Implementation

This chapter describes the solution on how to put all of the pieces (Database, Linux Containers, Openshift Platform, and Database replication) together and create an Operator to control and operate on top of all of this.

The database software used in the solution is MariaDB, which is described more in detail in Section 2.2.3. This decision was made based on the overall great performance and functionality of this database server. Also, MariaDB has remarkable documentation, which is really intuitive and user-friendly [27].

Another big impact on the decision was the existence of Galera replication software, which is built into the MariaDB server. More about Galera replication in Chapter 4. Galera replication provides an easy-to-configure environment and also a possibility of Galera Arbitrator daemon, which is described in Section 4.3.

The MariaDB server with a Galera replication software are bundled together into the container image. Section 5.2 describes the additional functionality that had to be added to the Red Hat's existing MariaDB server container images so this image can be later used by the operator in production. This section also describes Galera arbitrator container image creation which will be part of the Galera cluster managed by the operator.

In this case, the operator is responsible for managing the objects within the Openshift cluster. In case any of the deployed Pods crash, the operator has to control and handle such a scenario by redeploying another Pod with the correct parameters. These parameters are not always the same (they may change due to some circumstances like IP address changes), so it needs to keep track of it as well. Figure 5.1 helps to imagine how the operator interacts with the Openshift clusters objects.

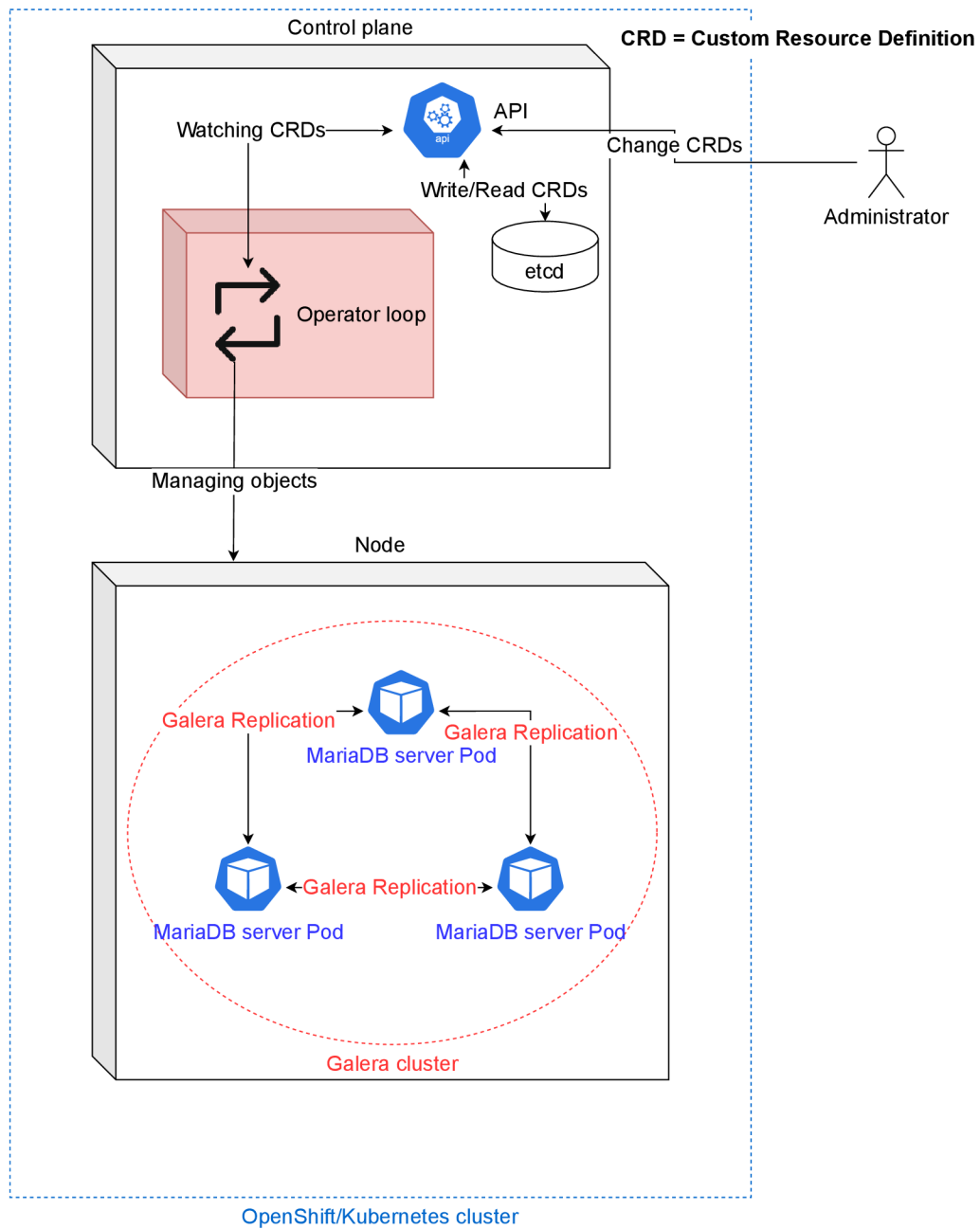


Figure 5.1: Simplified operator loop design inside of an OpenShift cluster. The operator manages objects like Pods and Deployments inside the cluster and watches (in loop) for the changes in the CRDs defined by administrator via Kubernetes API server. Whenever a CRD change occurs, the operator picks it up and applies it to the cluster.

The *Operator loop* design highlighted in Figure 5.1 is detailed in Figure 5.2. OpenShift operator implementation details are described in Section 5.7. Additionally, this section describes the earlier design attempts that led to the final, more robust one.

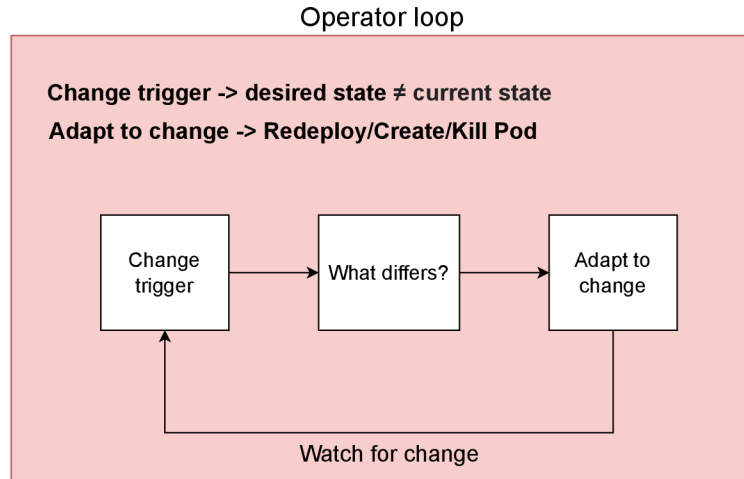


Figure 5.2: Operations inside the operator loop simplified to 3 simple steps. Operator watches for changes in the CRDs and in case that the desired state differs from current state, it adapts and manages targets to match the desired state. An operator also keeps an eye on the OpenShift cluster itself to ensure that it does not deteriorate (for example, a failed Pod) and perform fail-over recovery strategies if necessary.

At the end of this chapter, Section 5.8 wraps up the limitations of the designed MariaDB Galera OpenShift operator. These are derived from the used Galera replication solution as well as from the design decisions in the operator itself.

5.1 Existing solutions

Creating such a huge project from scratch consumes a lot of time and energy. It is therefore essential to look for some existing solutions that can be partly used as inspirations for the final solution.

5.1.1 PostgreSQL operator

Michal Cyprian, in his work, covers a similar solution with the use of a PostgreSQL database server [10]. The difference between MariaDB and PostgreSQL servers is quite huge, so the only part that could be observed as an inspiration was the OpenShift related solution. Additionally, the database replication in PostgreSQL is completely different from the one used in MariaDB.

Michal had to figure out and implement the Source-Replica promotions because the PostgreSQL replication is based on a single Source and multiple Replicas. It means that his operator has to deal with this issue and figure out which Replica is the best candidate for the next Source in the case that Source fails. In MariaDB Galera replication which is more described in Section 4.1, uses the Multi-Source architecture and does not have to do anything in case one of the Sources fails. The other Sources just update their Galera cluster nodes IP table and continue functioning.

5.1.2 MariaDB operator implemented by Operator community

The operator community has a dedicated Github repository¹ which is used to store the operator projects. Currently, there is one operator application created for the MariaDB database, but sadly it does not support Galera replication. This means that this operator setup can help only with the basic MariaDB configuration.

The community operator is also written in the Golang, which is used when the developer wants to have the full control over what the operator changes, however, in the case of this MariaDB Galera operator, it would not be necessary. Due to the fact that the Galera makes the replication quite easy to configure, the operator does not have to do much of a work on it. Therefore, the Ansible operator is sufficient in that case.

5.1.3 Ansible operator community

Ansible operator community provides a simple Ansible operator for tutorial² purposes. The main concept and functions of the Ansible-based operators shown in this example are useful for understanding how the Ansible-based operator works. Unfortunately, it does not cover too complex scenarios and strategies needed for the design of the MariaDB Galera operator. Ansible community also refers to their official operator Kubernetes module documentation³, which describes the Kubernetes modules that are supported by the Ansible-based operators.

5.2 Creating container image

Operator manages and deploys the container images, that had to be created at first. Sclorg⁴ has already created and maintains MariaDB container images. In other words, this solution can be used for further development and deployment in the OpenShift cluster. However, these images did not support and install the Galera software within themselves. Galera software is crucial for this thesis solution, so in order to use these container images, they had to be patched.

In the beginning, it was very important to comprehend how a container is created and what happens when it is started.

MariaDB container image⁵ supports multiple MariaDB server versions and also supports multiple operating systems (Fedora, RHEL, and CentOS). For the simplicity of this thesis, version 10.5 and the operating system Fedora was chosen as these are the most commonly used. Supporting other versions and operating systems is a topic for future development described in Chapter 7.

The next thing on the list was to edit the Dockerfile (described in Section 3.1.1). Additional packages like *galera.rpm* had to be added to the install section. For the OpenShift part (more about OpenShift in Section 3.4), new ports that are crucial for the Galera replication had to be exposed so the containers can communicate with each other in the cluster (these ports are described in Section 4.2).

During this process, a new problem within all Red Hat's Galera packages was discovered. Galera package had lacked the *procps-ng* package that is inevitable for Galera to work.

¹<https://github.com/operator-framework/community-operators>

²<https://sdk.operatorframework.io/docs/building-operators/ansible/tutorial/>

³<https://docs.ansible.com/ansible/latest/collections/kubernetes/core/>

⁴Container image group within the Red Hat.

⁵<https://github.com/sclorg/mariadb-container>

This issue was reported using the Red Hat's reporting system Bugzilla⁶, and later fixed by following the Fedora packaging guidelines [12] in the form of pull request⁷.

As containers do not allow to use of the systemd system control and service manager (*systemctl* [15]), the server has its own pre-phase, post-phase, and starting scripts. It was necessary to modify these scripts as well, since MariaDB needs to be started with Galera replication enabled in order to create a Galera cluster.

In order to start a stateful application, such as database, multiple variables have to be defined beforehand. These variables are passed to the database server through the Linux container environment variables. For example, it needs to know the user name and password in order to log in successfully. When adding a Galera into the equation, it needs to know the IP addresses, of the cluster nodes for the connection to work. Or in the other hand, if the Galera cluster is not yet created, the MariaDB server needs to know that, and start with a specific option to create the cluster. These Linux container environment variables are described below.

Galera control Linux container environment variables

- **GALERA_INIT**: This option is set without any value. When this option is set, the MariaDB server will initialize a new Galera cluster.
- **CLUSTERS_IP4**: This option is followed by one or multiple IP addresses separated by a comma. When this option is set, the MariaDB server will connect to the existing cluster with the connected nodes with the given IP addresses.

To pass these environment variables into the container, it is crucial to define them at the **run** command. Environment variables are defined using *-e* option passed to the *docker run* command. The following examples show what a docker run commands look like for the MariaDB Galera container.

Example of MariaDB Server container run command with Galera initialization:

```
$ docker run -e MYSQL_USER=user -e MYSQL_PASSWORD=pass \  
-e MYSQL_DATABASE=db -e GALERA_INIT= mariadb-galera-image
```

This command will initialize MariaDB Server with a user named *user*, which has password *pass* and creates a database named *db*. It also initializes a new Galera cluster.

Example of MariaDB Server container run command with connecting to the cluster:

```
$ docker run -e MYSQL_USER=user -e MYSQL_PASSWORD=pass \  
-e MYSQL_DATABASE=db -e CLUSTERS_IP4=10.11.0.3,10.11.0.5 \  
mariadb-galera-image
```

This command will also initialize MariaDB Server with a user named *user*, which has password *pass* and creates a database named *db*. However, this server will join an existing Galera cluster using the already connected nodes with IP addresses *10.11.0.3* and *10.11.0.5*.

⁶https://bugzilla.redhat.com/show_bug.cgi?id=2019805

⁷<https://src.fedoraproject.org/rpms/galera/pull-request/8>

5.2.1 Galera Arbitrator container image

Having the MariaDB container image with Galera replication supported created, the next step was to create a Galera arbitrator container image. This image was not part of the Sclog, so it had to be created from scratch. As decided at the beginning of designing this project, all of the containers will be used from Fedora Dockerfiles. This means that for the purpose of this project, only a Fedora container image had to be created. The development of other operating system container images is a topic for future work.

As mentioned above, to create any kind of container image, it has to start by creating a Dockerfile. The main idea behind containers is to minimize their size so they can be used in a low storage environment. That means the created container image consists only of the packages crucial for the Galera arbitrator to run properly.

After the Dockerfile was finished, the launching script has to be created next. Same as the MariaDB container image, the Galera arbitrator has to be started using a bash script. Fortunately, the Galera arbitrator has already a wrapper inside of its package. This helped with the IP addresses check and the configure file checks as well.

To ensure consistency across the container images, the **CLUSTERS_IP4** environment variable is used in the Galera arbitrator container as well. Arbitrators configure files have a different syntax compared to MariaDB Galera configuration files, so the IP addresses in the environment variable must use a different delimiter. It is necessary in this case to separate the IP addresses with a space (MariaDB uses commas).

Example of Galera arbitrator container run command with connecting to the cluster:

```
$ docker run -e CLUSTERS_IP4="10.11.0.3 10.11.0.5" garbd-image
```

This command will initialize the Galera arbitrator, which will automatically join an existing Galera cluster using the already connected nodes with IP addresses *10.11.0.3* and *10.11.0.5*. Galera arbitrator does not need any other environment variables to be defined, as it only works as a voter in the cluster and does not participate in replication. More about Galera arbitrator in Section 4.3.

Both of these container images are stored in a public Quay.io repository⁸. More about Quay.io in Section 3.4.6.

5.2.2 Create image streams for the new container images

All of the container images inside the Sclog repositories have their own image stream tags. Image streams have numerous advantages against just using a regular container images, which are more described in Section 3.4.2. In this case, two new image stream tags were added to the image stream JSON file. Sclog does maintain multiple image stream files for multiple operating systems. As a proof of concept, only one of these was edited.

The following listing is an example of a MariaDB server (with Galera support) **image stream** written in JSON format.

⁸<https://quay.io/user/ljavorsk>


```

1 {
2   "kind": "ImageStream",
3   "apiVersion": "image.openshift.io/v1",
4   "metadata": {
5     "name": "mariadb",
6     "annotations": {
7       "openshift.io/display-name": "MariaDB"
8     }
9   },
10  "spec": {
11    "tags": [
12      {
13        "name": "10.5-el8-galera",
14        "annotations": {
15          "openshift.io/display-name": "MariaDB Galera (RHEL-8)",
16          "openshift.io/provider-display-name": "Red Hat, Inc.",
17          "description": "Provides a MariaDB database on RHEL.",
18          "iconClass": "icon-mariadb",
19          "tags": "database,mariadb"
20        },
21        "from": {
22          "kind": "DockerImage",
23          "name": "quay.io/ljavorsk/mariadb-galera-init"
24        },
25        "referencePolicy": {
26          "type": "Local"
27        }
28      }
29    ]
30  }
31 }

```

More details about the image streams in OpenShift are described in Red Hat's official documentation [39]. There are multiple examples with a detailed description of how to use image streams throughout the OpenShift cluster.

All of these contributions are tracked in a fork project⁹ created from the original Sclorg MariaDB container project. The logically correlated sections are separated into the commits with a detailed description.

5.3 OpenShift API workflow

In order to work with the OpenShift cluster and manage its components, it is mandatory to understand how to get the commands to the cluster running in the cloud. There are two ways of doing this. OpenShift has made a long journey since its first launch, and nowadays, it is possible to use well developed and user-friendly web interface.

⁹<https://github.com/ljavorsk/mariadb-container>

However, every skilled developer should consider accessing the cluster through the OpenShifts API due possibility of a full control under the cluster. This could, and most of the time is, performed using the `oc` command (command-line tool using the OpenShift API to manage and operate on the cluster), which is described in Section 3.4 in much more detail.

There are plenty of subcommands inside of the `oc` command, which are worth mentioning and explaining because these are commonly used during OpenShift application development.

Nevertheless, nothing can be executed until the `oc` command does not know where to execute. This can be easily set using the `oc login` subcommand. This subcommand requires few options to work properly. One of them is `-server` which value specifies the address and port of the Kubernetes API server. The next option needs to specify the credentials mandatory to connect to the server. There are two ways to define those. One is to pass the value of `-user` and `-password`, which is not recommended to use because the password is stored in the bash library in plain text form. And the second way to pass credentials is to create and use a bearer token, also called a bootstrap token. More about this can be found in Kubernetes official documentation about authentication[16] Using the second option in OpenShift is very easy and also secure. It is even easier in the RedHat experimental OpenShift server called **QuickLab**.

5.3.1 Deployment server QuickLab

Creating and managing a custom OpenShift cluster or even server is quite expensive and time-consuming. There are so many configurations that need to be done first, and if a developer is new to OpenShift, they are going to struggle in at least some areas. Red Hat has come up with a perfect solution to this problem. It provides a simple, but under the hood, quite complex server. This server runs within *Red Hat's OpenShift Container Platform (RHOC)* and is referred to as the *QuickLab server*. Unfortunately, this server is available only inside of a Red Hat's intranet. Nonetheless, this server provides an unlimited custom OpenShift clusters for the developers that want their own space and do not want to share this space with anyone else. However, QuickLab also provides several shared clusters for the developers that do not need much space and want to only test their applications.

Using this shared cluster comes with a lot of preconfigured features and is a great place to start with developing a new application or just experimenting with the OpenShift cluster. Even though this cluster is shared, the applications are separated from each other in separate projects, so if a developer does not need to work with the whole cluster or the clusters configuration, using the shared cluster is the way.

Figure 5.3 shows how the QuickLabs Web UI looks with the exemplary MariaDB Galera project.

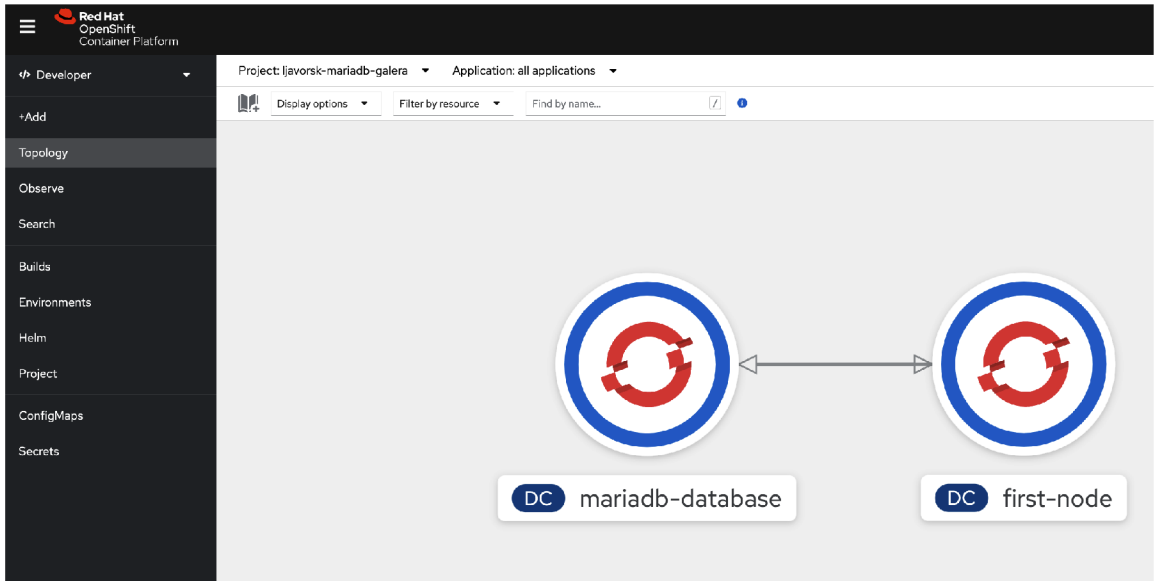


Figure 5.3: MariaDB Galera project displayed using QuickLab web UI.

Another great feature that QuickLab offers to its users is the *oc login* command generator. This generator generates the user-specific authentication bootstrap token, which is part of the *oc login* command mentioned above. QuickLab saves users even more time by generating the whole *oc login* command with every mandatory option included. An example of such QuickLab generated *oc login* command is written below.

Example of QuickLab generated *oc login* command:

```
$ oc login --token=<256_bit_token> \
--server=https://api.sharedocp4upi48.lab.upshift.rdu2.redhat.com:6443
```

In the interest of security, the token cannot be included, but its value is, for example, 256 bit hexadecimal number. This command connects OpenShift client to the shared cluster called **sharedocp4upi48**. The server uses a standard Kubernetes API server (TCP) port 6443.

5.3.2 Creating a new OpenShift project

The application needs to have its own space inside of an OpenShift cluster. To prevent applications from using or editing other applications resources, OpenShift divides them into separate projects. Projects serve as some kind of sandbox for the applications. The *oc new-project* command is then used to create a new project.

The project needs to have its own unique name and can (should) have a brief description and a display name that will be displayed in the UI. The developer can then switch between the projects using the *oc project <project_name>* command. If this command is used without the project name value at the end, it just displays the currently used project. More about the OpenShift projects in Section 3.4.1.

The following example shows how the *oc new-project* command works.

Example of creating a new OpenShift project:

```
$ oc new-project mariadb-galera --display-name="MariaDB Galera" \
--description="Development project for MariaDB server with Galera"
```

This command creates a new project called **mariadb-galera** with a display name **MariaDB Galera** and pretty self explanatory description.

5.3.3 Application deployment

The next thing after a successful login is to deploy an actual application. The application term serves to simplify all of the things Kubernetes needs to do to deploy, start and monitor the application.

For the developers that do not need to know what is really going on under the hood of deploying an application, OpenShift has created a simple *oc new-app* command. This command will automatically create a deployment config and replication controller and starts a service on top of the container. This service can be later exposed and accessed outside of the cluster. The replication controller then creates a Pod with the container image that is passed to the *oc new-app* command. The deployment config, replication controller, and service are more described in Sections 3.3.2 and 3.3.5.

There are several ways to pass the container image to the new-app command, for example, via a URL link to the public container registry, using the prestored container images from Red Hat, or using the custom image streams. The most commonly used option is the use of custom image streams because they have the easiest deployment and are great for version tracking. More about image streams in Section 3.4.2.

Following is an example of the *oc new-app* command in action.

Example of application deployment using oc new-app command:

```
$ oc new-app --image-stream=mariadb:10.5-el8-galera \
--name mariadb-database -e MYSQL_USER=user -e MYSQL_PASSWORD=pass \
-e MYSQL_DATABASE=db -e GALERA_INIT=
```

This command is using the image stream approach, with the custom image stream imported into the cluster. Importing files is described in the following Section 5.3.4. Image stream called **mariadb** with a tag called **10.5-el8-galera** is launched with several environment variables. These environment variables are mandatory for the MariaDB container to run, with exception of the **GALERA_INIT** variable. This environment variable is used to initialize the Galera cluster inside of the MariaDB server (more in Section 5.2). Container created by this command is then named **mariadb-database**, achieved by using the *-name* option.

5.3.4 Import custom files into the OpenShift cluster

As mentioned before, a lot of times, developers need to create their own files (image streams, deployment configs, ...) to address all needs of their specific applications. Creating such files is not a problem when following official OpenShift documentation [38][39].

However, the OpenShift cluster does not know where to find these files because it does not have access to the local computer. And so it is crucial to upload these files into the cluster using the *oc apply -f* command. These files are then stored in the resources of the cluster and can be used inside of it.

The following example commands show how to import these files into the cluster resources.

Example of importing the image stream and deployment config files into the cluster:

```
$ oc apply -f ./mariadb-rhel.json
```

```
$ oc apply -f ./mariadb-dc
```

The first command imports the image stream file called **mariadb-rhel.json** into the OpenShift clusters resources. And the second command imports the deployment config file called **mariadb-dc**. This file is a manifest for the MariaDB Galera server deployment configuration, used to deploy the MariaDB server into the Pod. Manifests are described in Section 3.3.8. After executing these two commands, the OpenShift client can refer to them in the following commands, as they are now part of the cluster's resources.

When a developer wants to check all of the image stream tags from an image stream, the *oc describe* serves the best. All tags from the MariaDB image stream can be displayed using the *oc describe is/mariadb* command. The *is* used in the command stands for image stream in this case.

5.3.5 Checking the status of projects components

When the crucial parts of the application deployment are deployed, the developer should always check if there is not something wrong. There are multiple ways of doing this, but all of them together are forming a great debugging toolset.

The best starting point for checking the status of the project is to just let *oc* command show it. The *oc status* subcommand serves to check and display the overall status of a project. Figure 5.4 shows the example of an *oc status* command's output.

```
[fedora@ljavorsk-fedora ~]$ oc status
In project ljavorsk-mariadb-galera on server https://api.sharedocp4upi49.lab.u
pshift.rdu2.redhat.com:6443

svc/first-node - 172.30.197.235 ports 3306, 4444, 4567, 4568
  dc/first-node deploys istag/mariadb:10.5-el8-galera
  deployment #1 deployed 41 hours ago - 1 pod

svc/mariadb-database - 172.30.183.22 ports 3306, 4444, 4567, 4568
  dc/mariadb-database deploys istag/mariadb:10.5-el8-galera
  deployment #1 deployed 41 hours ago - 1 pod
```

Figure 5.4: Current status of an exemplary OpenShift MariaDB project. This shows that there are two applications deployed (**mariadb-database** and **first-node**), from the same image stream tag **mariadb:10.5-el8-galera**. They both have only one Pod deployed and one service created on top of it.

Checking the Pods

As mentioned, the deployment config deploys a Pod. To understand what a Pod is, refer to Section 3.3.4. Nevertheless, when a problem within the cluster occurs, it is almost always problem within a Pod. That is why it is most important to know how to check it properly.

OpenShift client offers a `oc get` subcommand, which shows the details of a selected Kubernetes component. It could be deployment config, replication controller, or in this case, the Pod. To display the Pods status, the developer has to execute the `oc get pods` command.

Figure 5.5 shows the output of an `oc get pods` command and describes its content.

```
[fedora@ljavorsk-fedora ~]$ oc get pods
NAME                                READY   STATUS    RESTARTS   AGE
first-node-1-6k698                  1/1     Running   0           40h
first-node-1-deploy                  0/1     Completed 0           40h
mariadb-database-1-7hrgz            1/1     Running   0           40h
mariadb-database-1-deploy            0/1     Completed 0           40h
```

Figure 5.5: In the current project, there were four Pods deployed. Two of them were a deployment Pods that serve to deploy an actual running Pod. The running Pods (MariaDB servers) are called **first-node-1-6k698** and **mariadb-database-1-7hrgz**. From the status, it is obvious that they are still running without any issue.

The `oc get pods` command also has an option to display a more verbose status. This status includes also the IP addresses of each Pod, which will be important in the following sections when the Replication configuration comes into play. To display more information about the Pod, execute `oc get -o wide pods`.

Checking the deployment config or replication controller is not as commonly used as checking the Pods. However, the `oc get` command allows administrators to check both of them using the `oc get dc` and `oc get rc` respectively.

Getting the logs from containers running in the Pods

When a Pod crashes or cannot be deployed, it indicates that something must be wrong with the container inside of a Pod. In a local environment, the user can already see what is wrong thanks to the logs displayed from the container in real-time. However, when using the Kubernetes platform (such as OpenShift), there is no way to automatically display these logs to the user.

That is why the OpenShift client provides an `oc logs` subcommand. This command can be used on any resource within the project, but the most valuable logs are always in the Pods.

The example below shows how the `oc logs` command can stream the container logs from a Pod in real-time.

Example of streaming the logs from a Pod:

```
$ oc logs -f pods/mariadb-database-1-rxf6b
```

After executing this command, the real-time logs of a Pod named **mariadb-database-1-rxf6b** will be streamed into the current terminal session.

5.3.6 Service exposing

When creating a new application using the `oc new-app` command described in Section 5.3.3, it automatically creates a service on top of the Pods created to run this application. Service is an abstract way to expose an application running on a set of Pods as a network service. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods and can load-balance across them, so no resources are wasted [20]. More about a Kubernetes services is in Section 3.3.5

A freshly created service is not exposed by default, so if a developer wants to access the container outside of a cluster, it is impossible. However, the service can be *exposed*, which means that Kubernetes will create a Route to the service, which will expose it to the internet. This can be done using the `oc expose service <service_name>` command. Exposing a service on top of the MariaDB server will be crucial when it will be accessed outside of the cluster to be used in production.

5.3.7 Clusters port forwarding

The easiest way to connect, debug and work with a software is always on a local scale. In the case of the MariaDB server, this rule is the same. Fortunately, OpenShift provides an option to make this happen called *port forwarding*.

To get port forwarding from an OpenShift cluster to the local computer, a developer can use the `oc port-forward` command. This command requires two additional parameters. One is the name of the Pod, which should have its port/ports forwarded. And the second one is the remote port and the local port separated by a colon (e.g. 80:80).

The following example shows how the `oc port-forward` command works. And also how the MariaDB client can be connected to the server using a localhost domain.

Example of port forwarding the MariaDB server native port and connecting to the server via MariaDB client:

```
$ oc port-forward mariadb-database-1-rxf6b 3306:3306
```

```
$ mariadb -h '127.0.0.1'
```

This command will forward any local connections on port 3306 to the cluster's remote port 3306 of a Pod with the name **mariadb-database-1-rxf6b**. Port 3306 is the default MySQL/MariaDB protocol port. More about MariaDB in Section 2.2.3.

After successfully executing this command, the MariaDB client can be connected to the MariaDB server running in the OpenShift cluster, using the localhost domain. This can be accomplished by executing the second command in the example.

5.4 Custom deployment configs

Deployment config serves as a template for the Kubernetes cluster on how to deploy the application. The most common way is to write it in the YAML or JSON manifest. Section 3.3.8 describes what is a manifest, and Section 3.3.2 describes the deployment more into details.

When dealing with a stateful applications in container images, such as databases, it is important to launch them properly. The administrator needs to know a certain values before deploying the application.

In the MariaDB Galera container, the administrator needs to know the IP addresses of each cluster node before connecting to them. As mentioned in Section 5.2, this IP addresses are passed to the container image through the **CLUSTERS_IP4** environment variable.

Therefore, container images must be modified before they are deployed. The best approach for this in Kubernetes is to create a custom deployment configs.

5.4.1 MariaDB server deployment config

MariaDB container image has two possible deployment configurations. The first one is the case when the MariaDB Galera container is first deployed container, therefore has to initialize the Galera cluster (the container will be deployed with the **GALERA_INIT** environment variable).

And the other one is when the MariaDB server has to be connected to the existing cluster (the container has to be deployed with the **CLUSTERS_IP4** environment variable).

Either way, the deployment config has to be modified accordingly.

The following listing is an example of a simple configured MariaDB Galera server container deployment config. From the environmental variables, it is clear that the MariaDB server inside of this container will initialize the Galera cluster right after it is deployed (**GALERA_INIT** variable). It will also create a new database called **db**, create new user **user** with password credentials **pass**. Deployment config also exposes all of the mandatory TCP ports for the MariaDB and Galera software. These ports are described more in detail in Section 4.2. The image value had to be trimmed in this example due to its long hash number, which takes too much space.

```

1 {
2 apiVersion: apps.openshift.io/v1
3 kind: DeploymentConfig
4 metadata:
5   name: mariadb-database
6 spec:
7   replicas: 1
8   selector:
9     app: mariadb-database
10    deploymentconfig: mariadb-database
11  strategy:
12    type: Rolling
13  template:
14    metadata:
15      labels:
16        app: mariadb-database
17        deploymentconfig: mariadb-database
18        name: mariadb-database
19    spec:
20      containers:
21        - env:
22          - name: GALERA_INIT
23          - name: MYSQL_DATABASE
24            value: db
25          - name: MYSQL_PASSWORD
26            value: pass
27          - name: MYSQL_USER
28            value: user
29          image: "image-registry.openshift-image-registry.svc:5000/<hash>"
30          name: mariadb-database
31        ports:
32          - containerPort: 3306
33            protocol: TCP
34          - containerPort: 4444
35            protocol: TCP
36          - containerPort: 4567
37            protocol: TCP
38          - containerPort: 4568
39            protocol: TCP
40 }

```

5.4.2 Galera Arbitrator deployment config

Galera Arbitrator is easier to deploy because of the fact that it can only be connected to the existing cluster. It does not have the option to initialize the Galera cluster by itself.

In this case, the administrator has to determine the IP address of each node in the cluster and modify the **CLUSTERS_IP4** environment variable accordingly.

An example of a Galera arbitrator deployment config is shown below. When deployed, the Galera Arbitrator container will connect to the Galera cluster using the node with IP address **10.0.1.20**. Unlike MariaDB deployment config, Galera arbitrator does not need to expose the *3306* port because it does not participate in the actual replication.

```
1 {
2   apiVersion: apps.openshift.io/v1
3   kind: DeploymentConfig
4   metadata:
5     name: garbd
6   spec:
7     replicas: 1
8     selector:
9       app: garbd
10    deploymentconfig: garbd
11   strategy:
12     type: Rolling
13   template:
14     metadata:
15       labels:
16         app: garbd
17         deploymentconfig: garbd
18         name: garbd
19     spec:
20       containers:
21         - env:
22           - name: CLUSTERS_IP4
23             value: 10.0.1.20
24           image: "image-registry.openshift-image-registry.svc:5000/<hash>"
25           name: garbd
26           ports:
27             - containerPort: 4444
28               protocol: TCP
29             - containerPort: 4567
30               protocol: TCP
31             - containerPort: 4568
32               protocol: TCP
33       triggers:
34         - type: "ConfigChange"
35     revisionHistoryLimit: 2
36 }
```

5.5 Determining the cluster node IP addresses

When a Galera cluster is initialized, the first node's IP address is well known because it is easy to just check the first Pod's IP address. The administrator can use the *oc get* command described in Section 5.3.5 for this action. Once this is done, other Pods can be connected

to the cluster using the first Pod's IP address, and the connection should work without any issues.

However, the real challenge becomes when the first initial Pod crashed and needs to be restarted. The problem is that it cannot use the IP address of the first Pod anymore. That is the reason why the administrator has to always update the table of a connected cluster nodes. Thanks to the clever Galera replication architecture, it does not matter which node is used for connecting to the cluster, it just had to be a node already connected to it. More about the Galera replication architecture in Section 4.1.

Therefore, in such a case, an administrator must follow these steps.

- **Remember the connected nodes' IP addresses:** Whenever a node is connected to the cluster, the administrator has to remember the IP address of this Pod so they can refer to it in the future. In the case that the initial node fails and has to be restarted, the remembered IP addresses can be used to connect any new nodes to the cluster. In this case, a simple key-value table will suffice.
- **Check all other nodes connected in the cluster:** If another node's IP address has to be used to connect a new node, it is always good to check if this node is still part of the cluster. It may happen that this node has also crashed and cannot be used as a connection node. To check if this node is part of the cluster, the administrator has to connect to the MariaDB server and execute an SQL query (SQL is described in Section 2.2.2). Unfortunately, the MariaDB server does not store information about nodes' IP addresses and only stores their names. So the administrator has to check the name of the node and find out its IP address using the *oc get* command. The exact form of both the SQL query and the *oc get* commands are shown below.
- **Connect new node to the correct IP address:** After determining the correct IP address of a connected node, it can be used in the `CLUSTERS_IPV4` environment variable to connect to the cluster. Connecting to the cluster is more described in Section 5.2.

SQL query and *oc get* command used to determine the IP address of a connected node

```
$ mariadb -h 127.0.0.1 -uroot -e \  
"SELECT node_name FROM mysql.wsrep_cluster_members" \  
-s --skip-column-names  
  
$ oc get pod/<node_name> -o template \  
--template '{{.status.podIP}}{{"\n"}}'
```

The SQL query displays all clusters members' names in the plain text format, thanks to the used options. These names can be later used in the *oc get* command, which will display the exact IP address of the node in a clear plain text format.

5.6 Using Kubernetes Service IP address

While it is possible to communicate between pods using the IP addresses hard-wired to them as described in Section 5.5, it is not the most efficient way. The most pragmatic way

is to use the *Kubernetes Services* and its IP address. To understand Services in Kubernetes, refer to Section 3.3.5.

The biggest downside of using the Pod's IP address is that whenever the Pod dies, its IP address is re-used. It is re-used even in the case of upgrade or redeploy of the Pod. The Service can be bound to the Pod, and whenever the Pod is restarted, the IP address of a Service persists. This behaviour is especially useful in the case of replication. As a result, the Operator does not have to check IP addresses every single time some Pod dies, but rather just once during the initialization process.

5.7 Automating steps with Operator

The first step in creating an operator is to understand what the OpenShift operator is. For more information about operators, refer to Section 3.4.3. Moreover, it is important to know what the designed operator is supposed to do. Before automating the steps that need to be automated, the administrator should perform them manually. Using this method, the developer can easily configure what the operator should create, watch, and update, as well as set the targets of the operator. The method is similar to watching the administrator operate the cluster and automating his actions in the operator to reduce the organization's expenses.

In case of operator created for this project, the manual steps performed before automating them in the operator are described throughout the Sections 5.3 and 5.6. This operator's primary role is to monitor and create MariaDB Galera Pods with the correct Galera cluster control container environment variables. For the MariaDB server to communicate with the correct Galera cluster, the operator should deploy the MariaDB Galera container with the correct IP addresses of the Galera cluster nodes. With the Kubernetes Service (described in Section 5.6), in the case of the fail-over recovery, the operator does not need to verify that the IP address is correct, because the Service takes care of it. The operator then simply redeploys the Pod, and it will connect to the Galera cluster automatically.

5.7.1 Scaffolding operator project

Utilizing the *operator-sdk* tool, outlined in Section 3.4.4, eases the development phase of the operator. The operator-sdk automates a lot of OpenShift operator files that are typically the same for every project.

The operator used in the solution for this project is the so-called *Ansible-based* operator. Basically, the Ansible playbooks or roles are used to specify how the operators should behave. More about Ansible in Section 3.4.5. To generate the Ansible-based operator using the *operator-sdk* command, the *-plugins ansible* option must be defined. Ansible-based operators and API manifests built for Kubernetes can be scaffolded using the following commands.

Operator-sdk scaffold and API create Ansible-based operator

```
$ operator-sdk init --domain example.com --plugins ansible
$ operator-sdk create api --group cache --version v1alpha1 \
--kind Mariadbgalera --generate-role
```

After executing the commands listed above, the operator-sdk creates a fully functioning operator that is ready to be deployed. The operator does not have any logic at this point,

so it will not do anything, but it can be deployed in the cluster. The operator will create its own namespace in which it deploys the manager deployment. The manager deployment is basically the operator's controller used to manage the objects in the cluster. Operator-sdk describes these first steps in creating an operator in their tutorial [30].

5.7.2 Creating logic for the operator

Ansible-based operators allow developers to choose between two deployment approaches. The first consists of the well-known Ansible playbooks, and the other is a set of roles. These two options are nearly identical. In both cases, resources are handled in the same way using the YAML format. In contrast, the roles provide a few extra predefined variables that can be used in the manifests, as well as allowing easier roles sharing among users. A role approach is generally preferred to its additional values, however, playbooks are widely used in Ansible operator projects by developers that already have their playbooks predefined from other projects.

When the operator is deployed, it starts its own controller and manager used to interact with the OpenShift cluster resources. Additionally, a separate namespace with a name **mariadb-galera-operator-system** is created within the cluster. This namespace will become the aggregation point for all resources created by this operator.

Start-up deployment of the applications

After the operator is deployed and running inside of the OpenShift cluster, it immediately starts the watching loop for the *MariadbGalera* CRD. To understand definition of CRD refer to Section 3.3.7. When this CRD is created in the cluster, the operator picks it up and starts the initial deployment strategy.

The very first Pod which is deployed by the operator is the Pod with the predefined *GALERA_INIT* container environment variable (will be referred as **Main Pod** for the rest of this section). The container inside of this Pod creates the Galera cluster, so the other Pods can connect to it. To understand how this is achieved refer to Section 5.2. To ensure the Galera cluster is created before other Pods tries to connect to it, the operator waits until the *Main Pod* is deployed and running.

Operator then creates a Service used to encapsulate the Pods that are part of the Galera cluster, so all of them appears to have the same IP address in and outside the cluster. To understand what a Service is, follow Section 3.3.5. The Service then uses the built-in load-balancer to load-balance the network traffic across the Pods connected to it.

When the *Main Pod* is running, it means that the Galera cluster is initialized, and the other MariaDB Galera server Pods are ready to be deployed and connected to it. Operator creates a Deployment with user-defined number of replicas. This number is defined in the CRD watched by the operator. After the Deployment manages to deploy the Pods, the operator uses a custom defined health checks to check if these Pods are successfully connected to the Galera cluster and can start receiving the network traffic from the Service created previously. After the Pods pass the health checks, they become part of the Service as the Service endpoints.

Operator also creates a Deployment used to deploy the Galera Arbitrator Pods into the OpenShift cluster. Galera Arbitrator Pods then connect to the Galera cluster via the same Service as the MariaDB Galera server Pods.

Upon reaching the desired state of the cluster, the operator will terminate the *Main Pod* because it served its purpose of initializing the Galera cluster, but has no further value at this point.

Reacting to the failing Pods

The MariaDB Galera operator can detect the failure of a managed Pod and immediately respond with the fail-over strategy. It uses the Kubernetes Deployment defined fail-over strategies, so whenever a Pod managed by a Deployment dies, the Deployment will redeploy another Pod to replace the failed one. This fail-over strategy happens in a matter of seconds, so the user or the cluster administrator does not even notice it.

Custom Health checks

Another fail scenario can happen inside of the MariaDB Galera container image. When a MariaDB server inside of the container begins the process of connecting to the Galera cluster and it fails due to some network issues, the container stalls in the non-ready state and will not be restarted by the default Kubernetes behaviour. That is why the Kubernetes provides the *livenessProbes*. The *livenessProbes* are custom-defined health checks that can detect application-specific failure states. In the case of the MariaDB Galera container, the operator checks for the Galera connection by retrieving information from the Galera table record stored inside of the MariaDB database. This table is filled with information about the cluster only after the MariaDB server is actually connected to the Galera cluster.

MariaDB Galera operator implements one more custom health check (*readinessProbe*) defined for the MariaDB Galera containers. This health check was already mentioned above with the Service creation. Prior to connecting the MariaDB Galera servers to the Service, they need to be checked first. This check is performed on individual containers, and the purpose of this check is to test if the MariaDB server has been successfully connected to the Galera cluster, thus can become one of the Service endpoints used for the Galera network traffic.

Figure 5.6 depicts how the Service endpoints are connected to the Service only after they are labeled as *Ready*, which is handled by the *readinessProbe* custom health check.

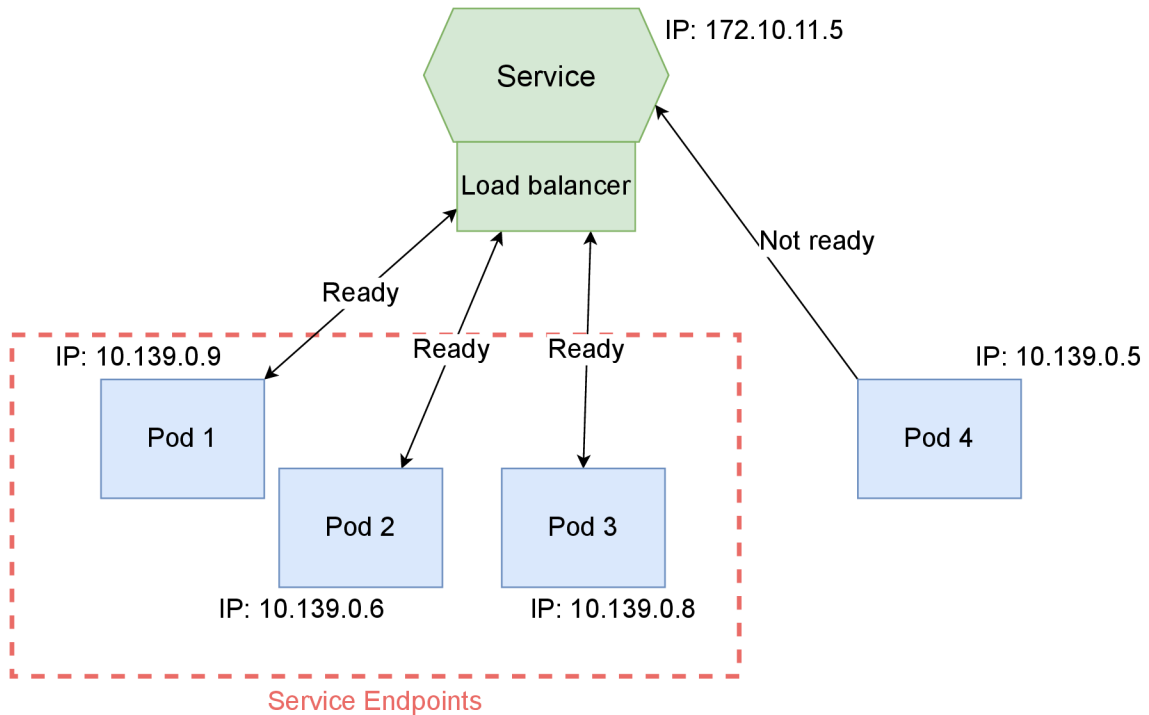


Figure 5.6: Network load-balancing solution used in the MariaDB Galera operator using the Kubernetes Services and custom health checks for Pod readiness. In this example the **Pod 4** is the only one not ready yet, and thus not part of the Service endpoints.

5.7.3 Deploying operator into the OpenShift cluster

In order to have the operator up and running, it is crucial to have the OpenShift cluster already configured. For testing and development purposes, the local OpenShift cluster is ideal due to its fast connection and ability to be configured quickly. Fortunately, Kubernetes provides multiple tools to create such a cluster. Example of such local cluster tool is *Minikube* or *Kind*, which are described in Sections 3.3.9 and 3.3.10 respectively. Local cluster solutions may struggle with networking issues, so the operator should be tested on the real OpenShift cluster prior to the production deployment.

As a consequence of the heavy Makefile-driven approach of the *operator-sdk* tool used to generate the basics of the operator, deploying the actual operator comes down to a few *make* commands. After having the OpenShift cluster connected with an admin permissions, the single **make deploy** command deploys the operator into the cluster. More about the operator deployment can be found in its Github repository¹⁰.

This command also creates a sample CRD manifest watched by the operator. After the CRD is applied in the cluster, it triggers the operator to start deploying the desired number of Pods in the OpenShift cluster. The sample of the CRD is showed in the listing below. The objective of this sample is to have five MariaDB Galera servers deployed and connected to the Galera cluster, along with two Galera Arbitrators.

¹⁰<https://github.com/ljavorsk/mariadb-galera-operator>

```
1 {
2   apiVersion: ljavorsk-mariadb-operator.com/v1alpha1
3   kind: MariadbGalera
4   metadata:
5     name: mariadbgalera-sample
6     namespace: mariadb-galera-operator-system
7   spec:
8     size: 5
9     garbdsiz: 2
10 }
```

5.8 Known limitations

The MariaDB Galera operator has some limitations when it comes to the very unlikely corner cases.

One of them is caused by the Galera replication software. This problem is described as “Two node limitation” [9], which is caused by the multi-master database replication design. This limitation is hit when there are no more than two nodes left in the Galera cluster. In a two-node cluster, when there is a conflicting pair of transactions, it could lead to split-brain situation. Conflicts can arise when both databases change the same table at the same time with no third node to decide which change to apply. The default cluster size defined in the operator has five MariaDB Galera server pods and two Galera Arbitrator pods to avoid hitting this limitation. Also, the extremely fast fail-over strategies defined in the operator help to keep the expected number of Pods alive.

The operator itself also has one limitation from the design. This limitation is, however, desired and expected. When all of the Pods in the OpenShift cluster die at the same time, the Galera cluster stops existing, and the OpenShift cluster is left with the Pods that cannot connect to it. The OpenShift cluster was designed to remain in its current state, so the administrator could examine the logs and determine the root cause of the error. The operator, by design prevents this limitation from happening by fast redeployment of the failed Pods. However, if this scenario happens and the logs are properly examined, the administrator can execute a simple **make redeploy** command, and it will automatically restart the operator with all of the OpenShift cluster Pods managed by it.

Chapter 6

Testing and evaluation

In order to be declared a valid piece of software, all new software should be properly tested to ensure its functionality. For this project, the testing consists of two parts, which are, respectively, targeted at different areas.

Test 1 is aimed at testing the individual container images. In order to measure container performance, Linux containers must work in the local environment first. And a second test will assess the performance and capabilities of the OpenShift Operator, as well as its failover and deployment strategies.

6.1 Local container testing

The container images must be tested before they can be used inside the OpenShift platform. Performing an initial startup test and testing the Galera cluster connection must both be satisfied.

The test that triggers all of the possible scenarios in the Galera cluster and checks that everything works as expected has been created in the fork¹ project of the MariaDB container image.

This test creates the container/podman network from where IP addresses will be obtained by the Linux containers. Then it starts the first initial MariaDB container, which initializes the Galera cluster. After the cluster is up and running, the test automatically starts the Galera Arbitrator, which connects to the freshly created Galera cluster. The next one on the queue is the MariaDB container which serves as a second node in the Galera cluster. After the second MariaDB server node is connected, the test shuts down the first initial MariaDB container to test if the cluster can (and should) process this change. When this is tested, it connects another Galera Arbitrator to check if the connection to the second (now primary) MariaDB node is successful.

After these tests are performed, the test moves to the clean-up phase, so there are no leftovers after the test ends. Also, the docker/podman network is removed, so it does not block the IP addresses that are not used anymore.

The test stores all the results and prints a useful messages during the process, so the user knows exactly what is being tested at the moment.

At the end of the test, the summary and the overall result of the test are printed to the standard output. If all of the tests (7 in total) passed, the result of the test is labeled as *PASSED*. If any of the tests fails, the overall result is displayed as *FAILED*. Figure 6.1

¹<https://github.com/ljavorsk/mariadb-container/blob/master/test/simple-deployment-test.sh>

shows how the test is displayed when executed on a local machine with the *docker/podman* software installed.

```
###  SETUP  ###
Cleaning the junk from the previous tests
Starting the MariaDB container used for testing the connection to servers ...

###  TESTING  ###
1: Testing the initialization of the galera cluster ... [ PASSED ]
2: Testing the cluster size after garbd connects to the cluster ... [ PASSED ]
3: Testing the connection of another mariadb-server to the cluster ..... [ PASSED ]
4: Testing removing garbd from the cluster ... [ PASSED ]
5: Testing removing initial mariadb server from the cluster ... [ PASSED ]
6: Testing the connection of new mariadb-server to the cluster ..... [ PASSED ]
7: Testing the connection of new garbd to the cluster ... [ PASSED ]

###  RESULTS  ###
7 / 7 passed

###  CLEANUP  ###
Stopping images:
mariadb_database2
mariadb_database3
mariadb_testing
galera_arbitrator

Removing docker network:
galera_cluster_network
```

Figure 6.1: Local deployment test ran in the fresh new Fedora machine. Each phase of the test is visually separated, so it is easy to navigate through it. The blue color in the tests messages highlights the important part of the explanation of what is being tested at the moment. The green and red colors are easier to spot, and they represent the result of the test.

This test uses the separate MariaDB container to connect the client to the currently tested MariaDB servers. When the connection is established client queries the server to check the Galera cluster values. To determine if a node has connected to the cluster, it uses the *wsrep_cluster_size* status variable. This variable represents the number of connected clusters. For the sake of simplicity of this test, IP address checking is not included. Either way, this test creates its own private VLAN where only the containers (nodes) created by the test can be connected.

6.2 Testing the operator

Operator should be always checking the status of every object in the Kubernetes/OpenShift cluster it is responsible for. These tests due to complexity of a Kubernetes deployments are performed partly manually.

After having the operator deployed and running, tests to check multiple fail-over scenarios can be performed. These tests were performed on the fresh new OpenShift cluster provided by the *Red Hat OpenShift Container Platform (RHOCP)*. Connecting to this type of cluster is covered in Section 5.3.1.

6.2.1 Desired state change adaption test

Considering the operator should watch for any changes done by the administrator in the Custom Resource Definition (CRD), it is definitely worth testing if the change in the CRD results in unexpected behavior.

Test has been performed executing patch commands, that changes the desired number of the MariaDB Galera servers and Galera Arbitrators in the OpenShift cluster. Examples showed below changes the CRD using the *oc* OpenShift client command.

Example of patching the CRD stored in the OpenShift cluster

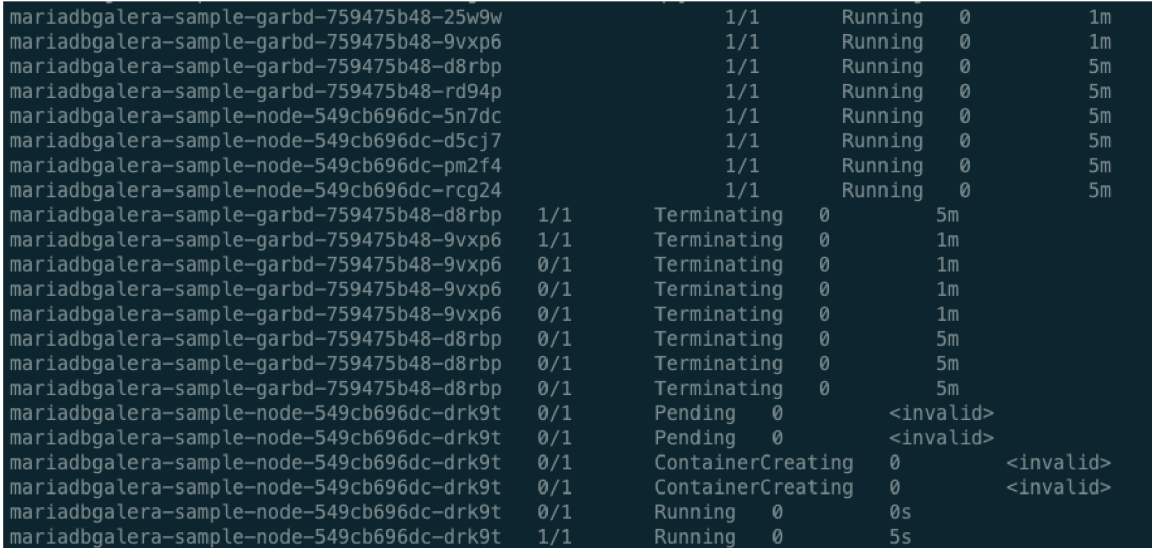
```
$ oc patch mariadbgalera mariadbgalera-sample \
-p '{"spec":{"size": 5}}' --type=merge \
-n mariadb-galera-operator-system

$ oc patch mariadbgalera mariadbgalera-sample \
-p '{"spec":{"garbdsiz": 2}}' --type=merge \
-n mariadb-galera-operator-system
```

First example changes the desired number of MariaDB Galera servers in the OpenShift cluster. Operator picks up this change and compare it with the actual state of the cluster. According to the difference it decides if it has to terminate the running Pods (if the desired number is smaller than actual number of MariaDB Galera servers) or create more Pods (if the desired number is bigger).

Second example does the exact same action with the Galera Arbitrator Pods.

Figure 6.2 shows how the operator reacts to the change defined in the example above. The actual state of the OpenShift cluster before this change was four MariaDB Galera servers and four Galera Arbitrators.



mariadbgalera-sample-garbd-759475b48-25w9w	1/1	Running	0	1m
mariadbgalera-sample-garbd-759475b48-9vxp6	1/1	Running	0	1m
mariadbgalera-sample-garbd-759475b48-d8rbp	1/1	Running	0	5m
mariadbgalera-sample-garbd-759475b48-rd94p	1/1	Running	0	5m
mariadbgalera-sample-node-549cb696dc-5n7dc	1/1	Running	0	5m
mariadbgalera-sample-node-549cb696dc-d5cj7	1/1	Running	0	5m
mariadbgalera-sample-node-549cb696dc-pm2f4	1/1	Running	0	5m
mariadbgalera-sample-node-549cb696dc-rcg24	1/1	Running	0	5m
mariadbgalera-sample-garbd-759475b48-d8rbp	1/1	Terminating	0	5m
mariadbgalera-sample-garbd-759475b48-9vxp6	1/1	Terminating	0	1m
mariadbgalera-sample-garbd-759475b48-9vxp6	0/1	Terminating	0	1m
mariadbgalera-sample-garbd-759475b48-9vxp6	0/1	Terminating	0	1m
mariadbgalera-sample-garbd-759475b48-9vxp6	0/1	Terminating	0	1m
mariadbgalera-sample-garbd-759475b48-d8rbp	0/1	Terminating	0	5m
mariadbgalera-sample-garbd-759475b48-d8rbp	0/1	Terminating	0	5m
mariadbgalera-sample-garbd-759475b48-d8rbp	0/1	Terminating	0	5m
mariadbgalera-sample-garbd-759475b48-d8rbp	0/1	Terminating	0	5m
mariadbgalera-sample-garbd-759475b48-d8rbp	0/1	Terminating	0	5m
mariadbgalera-sample-node-549cb696dc-drk9t	0/1	Pending	0	<invalid>
mariadbgalera-sample-node-549cb696dc-drk9t	0/1	Pending	0	<invalid>
mariadbgalera-sample-node-549cb696dc-drk9t	0/1	ContainerCreating	0	<invalid>
mariadbgalera-sample-node-549cb696dc-drk9t	0/1	ContainerCreating	0	<invalid>
mariadbgalera-sample-node-549cb696dc-drk9t	0/1	Running	0	0s
mariadbgalera-sample-node-549cb696dc-drk9t	1/1	Running	0	5s

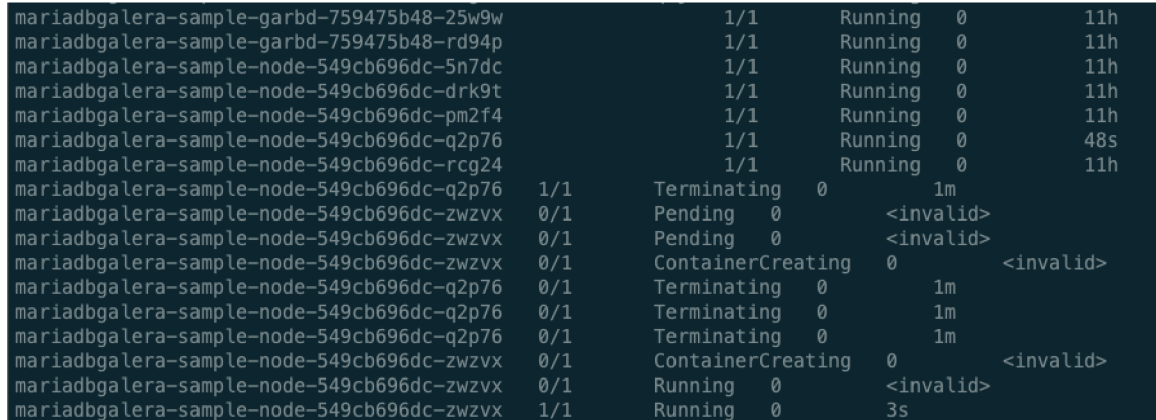
Figure 6.2: Live watching the operator-managed Pods after a change in CRD. In response to the change registered in the CRD, the operator terminated two Galera Arbitrator Pods and started one MariaDB Galera server Pod to match the desired state of the OpenShift cluster.

6.2.2 Fail-over strategy stress tests

MariaDB Galera operator is also capable of fail-over recovery strategies. These strategies ensure that any failed Pod is redeployed to ensure the application is not affected. It only takes a few seconds for the Pod to be redeployed, so there should be no observable difference for end users.

To test these strategies, OpenShift CLI offers a simple command (`oc delete pod`) that sends a SIGTERM signal to the Pod, which is selected for termination. Using this command on the running OpenShift cluster is a great way to stress test the fail-over recovery of the operator.

Figure 6.3 shows how the operator handles the simulated fail on one of the MariaDB Galera server Pods.



```
mariadbgalera-sample-garbd-759475b48-25w9w      1/1      Running    0          11h
mariadbgalera-sample-garbd-759475b48-rd94p      1/1      Running    0          11h
mariadbgalera-sample-node-549cb696dc-5n7dc      1/1      Running    0          11h
mariadbgalera-sample-node-549cb696dc-drk9t      1/1      Running    0          11h
mariadbgalera-sample-node-549cb696dc-pm2f4      1/1      Running    0          11h
mariadbgalera-sample-node-549cb696dc-q2p76      1/1      Running    0          48s
mariadbgalera-sample-node-549cb696dc-rcg24      1/1      Running    0          11h
mariadbgalera-sample-node-549cb696dc-q2p76      1/1      Terminating 0          1m
mariadbgalera-sample-node-549cb696dc-zwzvx      0/1      Pending    0          <invalid>
mariadbgalera-sample-node-549cb696dc-zwzvx      0/1      Pending    0          <invalid>
mariadbgalera-sample-node-549cb696dc-zwzvx      0/1      ContainerCreating 0          <invalid>
mariadbgalera-sample-node-549cb696dc-q2p76      0/1      Terminating 0          1m
mariadbgalera-sample-node-549cb696dc-q2p76      0/1      Terminating 0          1m
mariadbgalera-sample-node-549cb696dc-q2p76      0/1      Terminating 0          1m
mariadbgalera-sample-node-549cb696dc-zwzvx      0/1      ContainerCreating 0          <invalid>
mariadbgalera-sample-node-549cb696dc-zwzvx      0/1      Running    0          <invalid>
mariadbgalera-sample-node-549cb696dc-zwzvx      1/1      Running    0          3s
```

Figure 6.3: Operator fail-over recovery after SIGTERM signal sent to the **549cb696dc-q2p76** Pod. Operator automatically deployed new **549cb696dc-zwzvx** Pod to replace the failed one.

Operator is capable of handling multiple Pod terminations at the same time. This fail-over stress test is showed in Figure 6.4.

mariadbgalera-sample-garbd-759475b48-ng8q5	1/1	Running	0	56m
mariadbgalera-sample-garbd-759475b48-rd94p	1/1	Running	0	12h
mariadbgalera-sample-node-549cb696dc-5n7dc	1/1	Running	0	12h
mariadbgalera-sample-node-549cb696dc-drk9t	1/1	Running	0	12h
mariadbgalera-sample-node-549cb696dc-mdfpr	1/1	Running	0	56m
mariadbgalera-sample-node-549cb696dc-q9vpf	1/1	Running	0	56m
mariadbgalera-sample-node-549cb696dc-zwzvx	1/1	Running	0	1h
mariadbgalera-sample-node-549cb696dc-zwzvx	1/1	Terminating	0	1h
mariadbgalera-sample-node-549cb696dc-drk9t	1/1	Terminating	0	12h
mariadbgalera-sample-garbd-759475b48-rd94p	1/1	Terminating	0	12h
mariadbgalera-sample-node-549cb696dc-7hwp6	0/1	Pending	0	<invalid>
mariadbgalera-sample-garbd-759475b48-6psj9	0/1	Pending	0	<invalid>
mariadbgalera-sample-node-549cb696dc-7hwp6	0/1	Pending	0	<invalid>
mariadbgalera-sample-garbd-759475b48-6psj9	0/1	Pending	0	<invalid>
mariadbgalera-sample-node-549cb696dc-sjkbq	0/1	Pending	0	<invalid>
mariadbgalera-sample-node-549cb696dc-7hwp6	0/1	ContainerCreating	0	<invalid>
mariadbgalera-sample-node-549cb696dc-sjkbq	0/1	Pending	0	<invalid>
mariadbgalera-sample-garbd-759475b48-6psj9	0/1	ContainerCreating	0	<invalid>
mariadbgalera-sample-node-549cb696dc-sjkbq	0/1	ContainerCreating	0	<invalid>
mariadbgalera-sample-node-549cb696dc-drk9t	0/1	Terminating	0	12h
mariadbgalera-sample-node-549cb696dc-drk9t	0/1	Terminating	0	12h
mariadbgalera-sample-node-549cb696dc-drk9t	0/1	Terminating	0	12h
mariadbgalera-sample-node-549cb696dc-zwzvx	0/1	Terminating	0	1h
mariadbgalera-sample-node-549cb696dc-zwzvx	0/1	Terminating	0	1h
mariadbgalera-sample-node-549cb696dc-zwzvx	0/1	Terminating	0	1h
mariadbgalera-sample-node-549cb696dc-sjkbq	0/1	ContainerCreating	0	<invalid>
mariadbgalera-sample-node-549cb696dc-7hwp6	0/1	ContainerCreating	0	<invalid>
mariadbgalera-sample-garbd-759475b48-6psj9	0/1	ContainerCreating	0	<invalid>
mariadbgalera-sample-garbd-759475b48-rd94p	0/1	Terminating	0	12h
mariadbgalera-sample-garbd-759475b48-rd94p	0/1	Terminating	0	12h
mariadbgalera-sample-garbd-759475b48-rd94p	0/1	Terminating	0	12h
mariadbgalera-sample-node-549cb696dc-sjkbq	0/1	Running	0	<invalid>
mariadbgalera-sample-node-549cb696dc-7hwp6	0/1	Running	0	0s
mariadbgalera-sample-garbd-759475b48-6psj9	1/1	Running	0	0s
mariadbgalera-sample-node-549cb696dc-7hwp6	1/1	Running	0	4s
mariadbgalera-sample-node-549cb696dc-sjkbq	1/1	Running	0	4s

Figure 6.4: Second fail-over recovery stress test performed on the operator. After three Pods were simulated to fail simultaneously, operators can recover from multiple failures at once. In this test, two MariaDB Galera server Pods (**549cb696dc-zwzvx** and **549cb696dc-drk9t**), and one Galera Arbitrator Pod (**759475b48-rd94p**) were killed. New MariaDB Galera server Pods (**549cb696dc-sjkbq** and **549cb696dc-7hwp6**) and Galera Arbitrator Pod (**759475b48-6psj9**) replaced the failed ones.

6.3 Test results evaluation

Tests performed on the container images by creating multiple containers of MariaDB Galera server, and Galera Arbitrator with connecting all of them into the Galera cluster showed how robust and well designed the containers really are. This test, as mentioned in Section 6.1 is fully automated into the single bash script.

Simulated crashes in the MariaDB Galera operator’s managed cluster, as well as the tests dedicated to operator reactions to Custom Resource changes brought excellent results. These tests showed that the operator is capable of multiple simulated crashes at a time. In addition, the operator can achieve the administrator’s desired state within a very short time.

Table 6.1 provides an overview of the operator’s fail-over recovery time after multiple simulated crashes in the OpenShift cluster’s Pods.

Table 6.1: Operator fail-over recovery time. **Garbd** stands for Galera Arbitrator and **server** stands for MariaDB Galera server.

Cluster size (server/garbd)	Failed servers	Failed Arbitrators	Recovery time
4/4	2	1	5s
6/2	3	2	7s
5/3	1	3	6s
4/2	1	1	4s
8/6	5	3	10s
7/4	4	2	9s
5/2	2	2	5s

Similarly to the operator testing described in Section 6.2, these tests were also conducted on the Red Hat OpenShift cluster provided by RHOCP. All of the data was retrieved using the OpenShift CLI (*oc*) command, which is described more in detail in Section 3.4.

Chapter 7

Conclusion

The primary objective of this thesis was to design a system that can adapt and react to the current database server load by scaling and orchestrating a set of database servers in a cloud-like environment.

MariaDB database software was selected as a database system solution with a built-in Galera replication software for database replication traffic. MariaDB database servers form a Galera cluster, which connects all of its nodes together. These nodes could consist either from MariaDB database servers or Galera Arbitrators.

These applications had to be containerised in order to be independent from operating system and minimised for better portability. For these purposes, the Red Hat's MariaDB server container images were selected. MariaDB server container images from Red Hat's portfolio had to be patched for lacking Galera support, and for new Galera Arbitrator container image.

The scalability, orchestration, and deployment of the container images followed after the images were built and ready. OpenShift, Red Hat's enterprise-ready Kubernetes container platform was utilized for application deployment.

For generic applications, Kubernetes defined orchestration is sufficient, but every application has its own set of areas and events that need to be monitored. This is the reason why Kubernetes offers customizable operator solutions. The operator created for this thesis was based on Red Hat's automation enterprise framework called Ansible. The logic designed for the operator gives it the ability to detect any changes within administrator-defined custom resources, used to define the desired state of the OpenShift cluster, and immediately react to them. Depending on the change, either more containers are deployed to the OpenShift cluster, or superfluous containers are terminated.

Additionally, the operator is equipped with fail-over recovery strategies that can automatically detect a failing container and position a replacement in its place.

Numerous stress tests were conducted to demonstrate the operator's durability and quick response time in stressful situations like this. The operator proved to be capable of handling the hard workload pressure while coping with unfortunate scenarios in all of these tests.

Future work

As this operator was designed for the "proof-of-work" concept, it does not handle the persistent volume solution within the OpenShift cluster. Future work could try to implement

the use of OpenShift cluster persistent volumes in emergency situations, such as when the whole cluster is shut down.

Another plan for future development is the container image support for Red Hat Enterprise Linux (RHEL) as well as for the CentOS operating systems. For the purpose of research, only Fedora operating system images have been created.

Bibliography

- [1] BRENNAN, R. *Kubernetes Controllers Basics* [online]. February 2020 [cit. 2022-01-30]. Available at: <https://www.fairwinds.com/blog/kubernetes-controllers>.
- [2] BUTLER, T. *What is a Dockerfile?* [online]. June 2015 [cit. 2022-01-29]. Available at: <https://conetix.com.au/blog/what-is-a-dockerfile>.
- [3] CAREY, S. *What is cloud-native? The modern way to develop software* [online]. August 2021 [cit. 2022-01-27]. Available at: <https://www.infoworld.com/article/3281046/what-is-cloud-native-the-modern-way-to-develop-software.html>.
- [4] CASEY, K. *Kubernetes autoscaling, explained* [online]. March 2021 [cit. 2022-01-29]. Available at: <https://enterpriseproject.com/article/2021/3/kubernetes-autoscaling-explanation>.
- [5] CODECADEMY. *What is a Relational Database Management System?* [online]. [cit. 2022-01-24]. Available at: <https://www.codecademy.com/article/what-is-rdbms-sql>.
- [6] CODERSHIP LTD.. *Galera Arbitrator* [online]. [cit. 2022-02-07]. Available at: <https://galeracluster.com/library/documentation/arbitrator.html>.
- [7] CODERSHIP LTD.. *Replication API* [online]. [cit. 2022-02-07]. Available at: <https://galeracluster.com/library/documentation/architecture.html>.
- [8] CODERSHIP LTD.. *State Transfers* [online]. [cit. 2022-02-07]. Available at: <https://galeracluster.com/library/documentation/state-transfer.html>.
- [9] CODERSHIP LTD.. *Two-Node Clusters* [online]. [cit. 2022-05-07]. Available at: <https://galeracluster.com/library/kb/two-node-clusters.html>.
- [10] CYPRIAN, M. *High-availability for PostgreSQL in OpenShift* [online]. 2020. [cit. 2022-01-27]. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. Supervisor DOHNAL, V. Available at: <https://is.muni.cz/th/gx4ec/>.
- [11] DOCKER. *Dockerfile reference* [online]. [cit. 2022-01-28]. Available at: <https://docs.docker.com/engine/reference/builder/>.
- [12] FEDORA ORG.. *Fedora Packaging Guidelines* [online]. [cit. 2022-03-19]. Available at: <https://docs.fedoraproject.org/en-US/packaging-guidelines/>.
- [13] FOOTE, K. D. *A Brief History of Database Management* [online]. October 2021 [cit. 2022-01-23]. Available at: <https://www.dataversity.net/brief-history-database-management/>.

- [14] IBM. *What is database replication?* [online]. [cit. 2022-02-05]. Available at: <https://www.ibm.com/docs/en/i2-ibase/8.9.13?topic=replication-what-is-database>.
- [15] KERRISK, M. *Systemctl — Linux manual page* [online]. [cit. 2022-03-19]. Available at: <https://man7.org/linux/man-pages/man1/systemctl.1.html>.
- [16] KUBERNETES. *Authenticating in Kubernetes* [online]. [cit. 2022-03-30]. Available at: <https://kubernetes.io/docs/reference/access-authn-authz/authentication/>.
- [17] KUBERNETES. *Custom resources* [online]. [cit. 2022-02-13]. Available at: <https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>.
- [18] KUBERNETES. *Kind* [online]. [cit. 2022-05-07]. Available at: <https://kind.sigs.k8s.io/>.
- [19] KUBERNETES. *Minikube start* [online]. [cit. 2022-05-01]. Available at: <https://minikube.sigs.k8s.io/docs/start/>.
- [20] KUBERNETES. *Service* [online]. [cit. 2022-04-02]. Available at: <https://kubernetes.io/docs/concepts/services-networking/service/>.
- [21] KUBERNETES. *Horizontal Pod Autoscaling* [online]. December 2021 [cit. 2022-01-29]. Available at: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>.
- [22] KUBERNETES. *Kubernetes Components* [online]. October 2021 [cit. 2022-01-30]. Available at: <https://kubernetes.io/docs/concepts/overview/components/>.
- [23] KUBERNETES. *Operating etcd clusters for Kubernetes* [online]. November 2021 [cit. 2022-01-30]. Available at: <https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/>.
- [24] KUBERNETES. *Viewing Pods and Nodes* [online]. July 2021 [cit. 2022-01-31]. Available at: <https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/>.
- [25] KUBERNETES. *What is container orchestration?* [online]. July 2021 [cit. 2022-01-29]. Available at: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>.
- [26] KUBERNETES. *Kubelet* [online]. January 2022 [cit. 2022-01-31]. Available at: <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/>.
- [27] MARIADB. *MariaDB Server Documentation* [online]. [cit. 2022-02-10]. Available at: <https://mariadb.com/kb/en/documentation/>.
- [28] MARIADB. *Configuring MariaDB with Option Files* [online]. May 2013 [cit. 2022-01-26]. Available at: <https://mariadb.com/kb/en/configuring-mariadb-with-option-files/>.
- [29] MYSQL. *Overview of the MySQL Database Management System* [online]. [cit. 2022-01-25]. Available at: <https://dev.mysql.com/doc/refman/5.7/en/what-is.html>.
- [30] OPERATOR SDK. *Ansible Operator Tutorial* [online]. [cit. 2022-05-03]. Available at: <https://sdk.operatorframework.io/docs/building-operators/ansible/tutorial/>.

- [31] ORACLE. *Types of SQL Statements* [online]. [cit. 2022-01-25]. Available at: https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_1001.htm.
- [32] ORACLE. *What is Cloud Native?* [online]. [cit. 2022-01-27]. Available at: <https://www.oracle.com/cloud/cloud-native/what-is-cloud-native/>.
- [33] PEARCE, R. *Dead database walking: MySQL's creator on why the future belongs to MariaDB* [online]. March 2013 [cit. 2022-01-25]. Available at: https://www2.computerworld.com.au/article/457551/dead_database_walking_mysql_creator_why_future_belongs_mariadb/.
- [34] PERKINS, B. *Navigational Database* [online]. October 2010 [cit. 2022-01-23]. Available at: https://databasemanagement.fandom.com/wiki/Navigational_Database.
- [35] PERRY, Y. *Understanding Red Hat OpenShift Container Platform* [online]. July 2021 [cit. 2022-02-01]. Available at: <https://cloud.netapp.com/blog/cvo-blg-understanding-red-hat-openshift-container-platform>.
- [36] PHILIPS, B. *Introducing the Operator Framework: Building Apps on Kubernetes* [online]. May 2018 [cit. 2022-02-02]. Available at: <https://www.redhat.com/en/blog/introducing-operator-framework-building-apps-kubernetes>.
- [37] RAHIM, S. A. *Understanding the Kubernetes manifest* [online]. December 2019 [cit. 2022-02-04]. Available at: <https://medium.com/@sujithar37/understanding-the-kubernetes-manifest-97f44acc2cb9>.
- [38] RED HAT. *Deployments* [online]. [cit. 2022-03-30]. Available at: https://docs.openshift.com/container-platform/3.11/architecture/core_concepts/deployments.html.
- [39] RED HAT. *Managing image streams* [online]. [cit. 2022-03-22]. Available at: https://docs.openshift.com/container-platform/4.6/openshift_images/image-streams-manage.html.
- [40] RED HAT. *Red Hat Ansible Automation Platform* [online]. [cit. 2022-04-28]. Available at: <https://www.redhat.com/en/technologies/management/ansible>.
- [41] RED HAT. *What are Operators?* [online]. [cit. 2022-02-01]. Available at: <https://docs.openshift.com/container-platform/4.6/operators/understanding/olm-what-operators-are.html>.
- [42] RED HAT. *Working with projects* [online]. [cit. 2022-03-31]. Available at: <https://docs.openshift.com/container-platform/4.6/applications/projects/working-with-projects.html>.
- [43] RED HAT. *What's a Linux container?* [online]. January 2018 [cit. 2022-01-28]. Available at: <https://www.redhat.com/en/topics/containers/whats-a-linux-container>.
- [44] RED HAT. *What is container orchestration?* [online]. December 2019 [cit. 2022-01-29]. Available at: <https://www.redhat.com/en/topics/containers/what-is-container-orchestration>.

- [45] RED HAT. *Introduction to Kubernetes architecture* [online]. January 2020 [cit. 2022-01-30]. Available at: <https://www.redhat.com/en/topics/containers/kubernetes-architecture>.
- [46] SAMYAKJAIN5, SHREYASHAGRAWAL and NAVEENKUMARKHARWAL. *Relational Model in DBMS* [online]. June 2018 [cit. 2022-01-25]. Available at: <https://www.geeksforgeeks.org/relational-model-in-dbms/>.
- [47] SUMATHI, S. and ESAKKIRAJAN, S. *Fundamentals of Relational Database Management Systems*. 1st ed. Springer Publishing Company, Incorporated, 2010. ISBN 364208012X.
- [48] TECHNOPEdia. *Database Management System (DBMS)* [online]. August 2020 [cit. 2022-01-24]. Available at: <https://www.techopedia.com/definition/24361/database-management-systems-dbms>.
- [49] VERTICE. *Understanding Structured Query Language (SQL)* [online]. April 2020 [cit. 2022-01-25]. Available at: <https://verticecloud.com/understanding-structured-query-language-sql/>.
- [50] WICKRAMASINGHE, S. and RAZA, M. *DBMS: Database Management Systems Explained* [online]. bmc.com, 09. december 2021 [cit. 2022-01-23]. Available at: <https://www.bmc.com/blogs/dbms-database-management-systems/>.
- [51] XU, J. *Why Implementing Kubernetes Operators Is a Good Idea!* [online]. June 2021 [cit. 2022-02-10]. Available at: <https://www.kubermatic.com/blog/why-implementing-kubernetes-operators-is-a-good-idea/>.