MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Cloud Infrastructure Health Monitoring System

Master's Thesis

BC. JAN ŠAFAŘÍK

Brno, Fall 2021

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Cloud Infrastructure Health Monitoring System

Master's Thesis

BC. JAN ŠAFAŘÍK

Advisor: Mgr. Martin Večeřa

Department of Computer Systems and Communications

Brno, Fall 2021

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Jan Šafařík

Advisor: Mgr. Martin Večeřa

Acknowledgements

I would like to thank to my thesis advisor Mgr. Martin Večeřa and consultant Mgr. Tomáš Sedmík for their guidance and motivation. I would also like to express my gratitude to my family and friends for their support during both my studies and writing of this thesis.

Abstract

The aim of this diploma thesis is to analyze current solutions for health monitoring of multiple Red Hat OpenShift Container Platform (OCP) clusters. With consideration to the outcome of the analysis, design and implement a system for health monitoring of multiple OCP clusters deployed and used in an environment for quality engineering purposes. In addition to health monitoring, the system exposes an API providing information about current infrastructure status. The thesis also considers the cloud provider hosting the OpenShift clusters called Red Hat OpenStack.

Keywords

Docker, Containers, OpenShift Container Platform, OpenStack, Enterprise Kubernetes, Monitoring

Contents

1 Introduction **2**
- 1.1 Problem description . . . 2
  - 1.1.1 QE environment . . . 3
- 1.2 Thesis goals . . . 3
- 1.3 Thesis structure . . . 4

2 Health monitoring **5**
- 2.1 Overview . . . 5
- 2.2 Types of monitoring . . . 6
  - 2.2.1 White box monitoring . . . 6
  - 2.2.2 Black box monitoring . . . 6
- 2.3 Monitoring tools . . . 7

3 Images and Containers **8**

4 Kubernetes **10**
- 4.1 Architecture . . . 10
- 4.2 Tools . . . 11
- 4.3 Types of resources . . . 12
  - 4.3.1 Pod . . . 12
  - 4.3.2 ReplicaSet . . . 13
  - 4.3.3 ReplicationController . . . 15
  - 4.3.4 Deployment . . . 15
  - 4.3.5 Service . . . 17
  - 4.3.6 Ingress . . . 18
  - 4.3.7 Namespace . . . 19
  - 4.3.8 Persistent Volume Claim and Persistent Volume 20
  - 4.3.9 Node . . . 22

5 OpenShift **23**
- 5.1 Interaction tools . . . 23
- 5.2 Types of resources . . . 23
  - 5.2.1 Project . . . 24
  - 5.2.2 ImageStream . . . 24
  - 5.2.3 BuildConfig . . . 25

5.2.4 Build 26
5.2.5 DeploymentConfig 27
5.3 Common problems with OpenShift clusters in QE environment 27
5.3.1 Human error 28
5.3.2 Infrastructure instability 28
5.4 Current monitoring solutions 30
5.4.1 OpenShift Monitoring Stack 30
5.4.2 VictoriaMetrics 31
5.4.3 DataDog 31
5.4.4 Other open-source projects 31

6 OpenStack **33**
6.1 Infrastructure as a Service 33
6.1.1 Types of resources 33
6.2 Common problems 34
6.2.1 Resource cannot be deleted 34
6.2.2 Slowness 35
6.3 Current monitoring solutions 35
6.3.1 OpenStack telemetry 35

7 Proposed solution **37**
7.1 Requirements 37
7.1.1 Functional requirements 37
7.1.2 Non-functional requirements 37
7.2 Development stack 38
7.2.1 Java 38
7.2.2 Quarkus 38
7.2.3 Fabric8 Kubernetes client 39
7.2.4 Camel 39
7.2.5 Quartz 40
7.2.6 Prometheus 40
7.2.7 Grafana 41
7.2.8 OpenShift 41
7.3 Architecture 41
7.3.1 Sample application 41
7.3.2 Multi OCP Monitoring 43

8 Implementation **44**
8.1 Sample application 44
8.1.1 Design 44
8.1.2 Deployment 46
8.2 Multi OCP Monitoring 47
8.2.1 Design 47
8.2.2 Deployment 50

9 Conclusion **51**
9.1 Future work 51

Bibliography **52**

List of Figures

4.1 The components of a Kubernetes cluster(10) 11
7.1 The architecture of the proposed solution 42
8.1 Class diagram for the sample application 45
8.2 Class diagram for the Multi OCP Monitoring application 48

Abbreviations

CDI Context and Dependency Injection 39

CLI Command-line interface 26

DSL Domain specific language 39

JVM Java Virtual Machine 38

LTS Long Term Support 38

OCI Open Container Initiative 8

OCP OpenShift Container Platform 2, 3, 4, 23, 24, 25, 26, 27, 28, 29, 30, 33, 34, 35, 41, 42, 44, 46, 47, 49, 50, 51

PV Persistent Volume 20, 21, 22, 30, 34, 42, 46

PVC Persistent Volume Claim 20, 21, 22, 46

QE Quality Engineering 2, 3, 27, 50, 51

YAML YAML Ain't Markup Language 12, 14, 15, 17, 19, 20, 21, 24, 25, 46, 50

1 Introduction

Throughout the whole world, there is a massive growth in application containerization, cloud-native applications, and cloud services. Most companies are looking for an environment that would be stable and production ready, easing the process of deployment and ensuring the applications' smooth running. As the number of containerized software is rising, so is the need for an inexpensive and reliable testing environment to ensure the quality of the applications. Nowadays, the time between the build of an application and its public release is getting shorter and shorter, requiring the testing to be quick and precise. This brings the need to have multiple environments with common configuration always ready.

1.1 Problem description

The thesis' topic is cloud infrastructure health monitoring. Specifically, the focus is on health monitoring of Red Hat OpenShift Container Platform (OCP) installed on Red Hat OpenStack platform dedicated to Quality Engineering (QE) purposes. More information about both platforms will be provided later in this thesis.

Usually, QE teams need multiple instances of testing environments. There can be a lot of reasons for that, such as distribution of tests in case of an extremely high number of test cases, testing of multiple product versions simultaneously each in an isolated environment, so-called "playground" instances used by QE team members for new test development and more.

As the container platform is installed in space dedicated for QE, some of the OCP instances can stop behaving correctly or become inaccessible due to various reasons. For example, OpenStack maintenance or upgrade of some of the cloud provider's components can negatively affect already created resources. Similarly, a human factor is playing a big role which can lead to misconfiguration of OCP resulting in incorrect behavior.

To ensure that the team has enough instances ready to use, it is necessary to monitor the status of each instance. Thanks to monitoring it is possible to detect any issues with the current infrastructure and

respond accordingly. This can lead to testing stabilization, automatic destruction of failing instances, etc.

1.1.1 QE environment

As it has been already mentioned, the stack used for this thesis is Red Hat OpenShift Container Platform instances running on Red Hat OpenStack. The OpenStack configuration for QE is not vastly different from a production ready environment. Although the OpenStack configuration does not differ from the production environment much, the selected installation process of OCP can.

There are usually multiple OCP clusters running simultaneously for each product. Each cluster can have a different purpose. Examples of such purposes could be:

- testing of product patches
- testing of the latest unreleased version
- performance testing environment
- playground environment
- product upgrade testing

and many more. Some of those instances, for example the one dedicated to the testing of product patches, can be connected to testing pipelines automatically triggered by a new product build. These instances that are part of continuous integration pipelines need to be in the best shape possible, as there is no manual verification of the environment's health. On the other hand, some instances are used mainly by QE testers. Even though it may be easily discovered if a cluster is not responding, some issues may not be so easily spotted. This establishes the reason why it is important to have a health monitoring solution in place.

1.2 Thesis goals

The goal of this thesis is to find and analyze current solutions for monitoring of multiple OpenShift Container Platform clusters. The ideal

solution should be able to check the correct behavior of workload resources provided by OCP, track the number of currently installed and currently working clusters, expose an API with the ability to provide information about a cluster, its health status, and offer a possibility to integrate other platforms, such as the push of notifications to Google Chat or integration with CI tools such as Jenkins to automatically reinstall a not working cluster. With respect to the outcome of the analysis, design and implement a system meeting all previously mentioned requirements.

1.3 Thesis structure

As the thesis works with complex topics and systems, there is a thorough introduction to each area. Chapter 2, called Health monitoring, covers the fundamentals of application monitoring, lists different types of monitoring, and introduces some widely used tools helping with it.

Chapter 3, called Images and Containers, starts with the introduction to what a containerized application is and focuses on its essential terms. Chapter 4, called Kubernetes, builds on top of it by providing a detailed description of the orchestration tool and its workload resources. Finally, Chapter 5, called OpenShift, provides information about the platform extending Kubernetes and states the common issues QE teams encounter.

Chapter 6 is called OpenStack and similarly to chapter 5 it introduces the OpenStack platform, different types of resources provided by the platform, and common problems.

Chapter 7 and 8 are devoted to a newly proposed solution for OCP health monitoring. Chapter 7 covers the requirements, overall architecture, and development stack used for the monitoring system and chapter 8 covers the implementation details of each component, possible configuration, deployment, and usage.

Finally, chapter 9 provides a conclusion of this thesis together with possible future extensions.

2 Health monitoring

2.1 Overview

Application health monitoring is the process of checking various parameters of an application in order to estimate the status of the running system. Such monitoring can be as simple as information if a service is currently working or not, but can also include more detailed data, such as the current number of requests, the percentage of CPU usage, the remaining disk space, etc. According to *Effective Monitoring and Alerting* by Ligus the term *monitoring* can be interpreted in many ways and can represent many different processes and approaches.

Ligus further explains that the main advantage of monitoring is the ability to detect threatening issues as soon as possible. One of the commonly monitored aspects of a system is its availability. It simply provides information on whether the application is working and ready to be used. Another often monitored property is the performance of the application (1).

Finally, Ligus states that with the data provided by monitoring it is possible to determine the optimal values, and in case of abnormal values take reasonable action. It also allows predicting the future development of a given situation. When the values are out of the limits, it can result in regulation of the usage of the monitored system and should result in a repair. Both of these actions can be either manual or automated (1).

Monitoring should be an important part of most big projects. A lot of services and companies such as GitHub[1], Facebook[2], Digital Ocean[3] or Red Hat[4] publish results of their health monitoring online on their status pages. This helps other services to discover if everything is working so they can use the given functionality.

Even though the above-mentioned status pages are monitoring public running services, we can apply the same for any other running software, no matter if public, private, internal, or external.

---

1. Status page located at `https://www.githubstatus.com`
2. Status page located at `https://status.fb.com`
3. Status page located at `https://status.digitalocean.com`
4. Status page located at `https://status.redhat.com`

Thanks to this, it is possible to integrate various services together and be prepared for any possible outage coming from a third-party service.

2.2 Types of monitoring

Monitoring can be divided into different types based on various aspects. For example, availability and performance monitoring, white box and black box monitoring, etc. While availability and performance monitoring should be self-explanatory terms, white box and black box monitoring are not so common and are described in the following subsections.

2.2.1 White box monitoring

According to Ligus, the monitoring data collected by the collection agent is gathered from within the system. As an example, a white box monitoring agent could parse log files on a server or read CPU utilization (1). In other words, white box monitoring is enabled by internal access to the monitored system. This type of monitoring provides the most detailed view of the service.

2.2.2 Black box monitoring

Contrary to the white box approach, black box monitoring runs outside of the system. An example could be an application that is trying to send GET requests every minute to the monitored service's API and analyzing the returned response to estimate the health of the service (1).

This type of monitoring is helpful in cases when it is not possible to get the collection agent inside the system. Furthermore, with this approach monitoring of multiple systems and gathering the data on a single database is easier.

Additionally, it might be a good idea to combine both types of monitoring, a black box for the discovery of the overall accessibility of the service or system and a white box for more detailed and precise monitoring.

2.3 Monitoring tools

This section introduces some of the most popular tools helping with monitoring. This list is incomplete as there are many other solutions available and comparison of such tools is not the main goal of this thesis. Additionally, it is important to note that the tools mentioned in this section are not solutions providing mechanisms to monitor an application, but rather tools providing easy gathering, tracking, storing, and visualization of the application's status data. This means, that either the tracked application or other service monitoring the health has to provide current status data to the tool.

Prometheus is, according to their website documentation, an open-source monitoring solution originally developed at SoundCloud[5]. The tool has joined the Cloud Native Computing Foundation in 2016 to affirm the project's independence of any company. Prometheus is collecting and storing metrics as time-series data with optional labels (2). Each metric can be queried to get various statistics of the monitored service. The metrics data are pulled using an API endpoint exposed by the monitored service.

Graphite is collecting numeric time-series data and is able to render graphs using it (3). Unlike Prometheus, Graphite expects the monitored application to push the data to the monitoring tool (4). According to their website, some of the users of Graphite are Github, Salesforce, and Booking.com (5).

OpenTSDB is, according to their website, an open-source time-series database built using Apache Hadoop and Apache HBase. Similar to Graphite, OpenTSDB is expecting the monitored application to send the metrics data to the monitoring tool. After that, the data can be viewed either in its built-in user interface or using external monitoring dashboards (6).

5. `https://soundcloud.com`

3 Images and Containers

Let's start with a short summary of what an *image* and a *container* are and their associated terminology.

Container is, according to information provided by the Docker website, a running package of software with all of its dependencies, runtime, system tools, system libraries, and more. All containers are running on a selected container engine, and share the machine's OS system kernel, resulting in high efficiency. Thanks to this, we are able to start isolated prepared environments running uniformly on multiple machines with different OSes (7).

Container Image or just *image* is a lightweight, standalone and executable package (7). An image becomes a container when it runs on a container engine - a good analogy could be that an image is to a container what a class is to an object in object-oriented programming. An image consists of multiple image layers each representing changes between itself and the previous layer (8). To create a new *image*, one popular approach is to create a file containing all necessary steps for our final image. Such a file is usually called *Dockerfile* and many container tools are able to build a new image using it.

Container engine enables the interaction with images and containers (8). When installed, container engines usually contain a CLI for user interaction. Different container engines might support a different set of operations. Common and most core functionality is the ability to pull an image from an image registry server, start and stop a container, inspect a running container, and more (8). There are many container engine implementations, such as *Docker* or *Podman*.

Open Container Initiative (OCI) according to *FAQ - Open Container Initiative* aims to provide specifications and standards around container technology. It includes the format of images and containers to ensure smooth running across multiple container engines as well as operating systems. It also includes a standard set of actions that are possible to perform on both images and containers (9).

Registry server is a server holding container images. When an image is not present locally, most container engines will try to pull it from a predefined registry server. For example, when you try to start a container using Docker, but the image is not cached locally, Docker will try to pull the image from `docker.io`. Another example of a widely used public registry server is `quay.io`. Thanks to registries, it is possible to easily distribute and store created images. Although there are public services such as `docker.io` and `quay.io`, it is also possible to run image registry locally or on any server.

4 Kubernetes

As today's applications tend to move from a monolithic architecture to microservices, a lot of containerized software requires multiple containers running simultaneously. With most container engines this would require manual start and configuration of each container in order to start the whole application. Similarly, scaling an application by deploying multiple containers and their management requires higher effort. And there are other scenarios that might require the developer's or maintainer's attention, such as a container in an error state.

Kubernetes addresses all of the previously mentioned problems. It is an orchestration system used to manage the deployment of containers, their scaling, networking, and much more. It is important to understand the essentials of Kubernetes as the OpenShift Container Platform is built on top of it.

This section provides motivation and a description of some of the most fundamental and important parts of Kubernetes in order to establish an overview of the platform. Each subsection explains the basic idea, but is not trying to replace the official documentation, nor is addressing all aspects of the given topic. Additionally, this section does not cover all features of Kubernetes, as not every feature is critical to health monitoring.

4.1 Architecture

As shown in figure 4.1, the main components of a Kubernetes cluster are the Control plane and Nodes. The Control plane manages the Nodes, exposes the cluster API, etc. The Nodes, also commonly referred to as Worker nodes, are primarily hosting the containers (10).

A production Kubernetes cluster will usually consist of multiple machines. There should be more than one machine running the Control plane to keep the cluster running even if one of the machines fails. The number of Worker nodes should correspond to the intended workload.

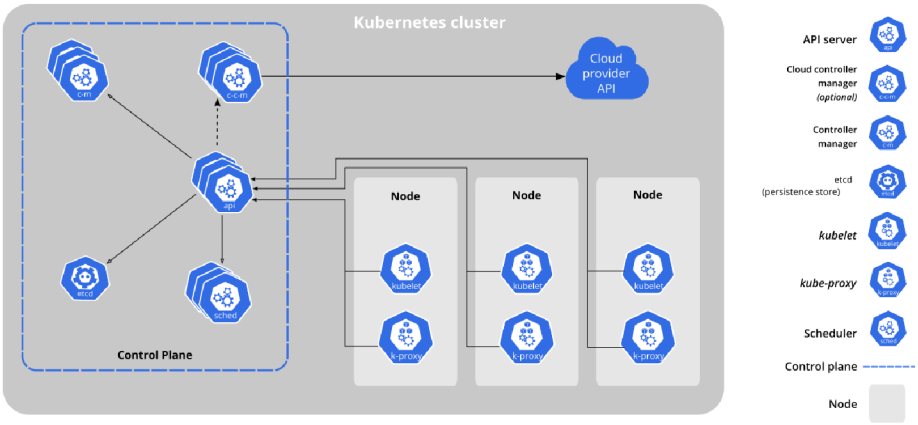

Figure 4.1: The components of a Kubernetes cluster(10)

Fortunately, we can also run a Kubernetes cluster for development, learning, or other purposes on a single machine. The easiest way to run such a cluster is to use Minikube[1].

4.2 Tools

The most necessary tool required for this section is *kubectl*. It is a CLI used for quick and easy interaction with a Kubernetes cluster, enabling actions such as application deployment, an inspection of cluster resources, connecting to running containers, etc.

It is important to configure *kubectl* to communicate with the Kubernetes cluster. The tool stores information about all clusters, users, and contexts linking the two together in a config file usually located at *~/.kube/config* (on Linux machines).

There are many other tools, each supporting a different aspect of work with Kubernetes. For example, *kubeadm* for easy install of Kubernetes cluster, or already mentioned *minikube* for easy setup of single node cluster for development purposes. More tools can be found in the documentation of Kubernetes on their official website[2].

1. More info about Minikube can be found on their website: `https://minikube.sigs.k8s.io`
2. `https://kubernetes.io/docs/tasks/tools/`

4.3 Types of resources

This section covers some of the primary resource types that a user is able to interact with inside a Kubernetes cluster. A resource is an object inside Kubernetes which represents given functionality.

This section is not listing all resource types, as some of them are not important for this thesis.

Most resource type subsections include an example snippet in *YAML Ain't Markup Language* (*YAML*) format, which could be used to deploy a given resource. One way to create the resources specified in the snippet is to save it into a file, let's call it *resource.yaml*, and execute the command `kubectl create -f resource.yaml`.

4.3.1 Pod

"Pods are the smallest deployable units of computing that you can create and manage in Kubernetes." (11)

A Pod is the most basic and essential resource, as it is the resource representing a container. To be correct, it is possible to run more than one container inside a Pod, but the most common use case is to run only one container per Pod(12).

Let's examine the following YAML representation of a Pod:

Listing 4.1: Pod snippet in YAML format

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: grafana
  name: grafana
spec:
  containers:
  - image: grafana/grafana:latest
    imagePullPolicy: IfNotPresent
    name: grafana
    ports:
    - containerPort: 3000
```

As you can see in the snippet 4.1, each Pod has its name and possible labels (located in *.metadata*). In this case, the Pod's name is *grafana*, and it has one label - *app: grafana*.

The most interesting part is the specification of the container running inside the Pod. In this example, there is a *grafana* image with the tag *latest* used. If such an image is not present in the local Kubernetes image registry, it should be pulled from a public registry server (this behavior is defined on line 10). Additionally, you can see that port *3000* is exposed, which can be later used to connect to this pod/container.

When this resource is deployed, the cluster should create a new Pod, find the container image locally or in a public registry server and start the container.

4.3.2 ReplicaSet

Motivation

Direct deployment of a Pod does not provide many advantages. When multiple Pods with the same container and configuration need to be deployed, it would have to be done manually. Even though the deployment of many Pods could be automated with a script[3], it would only provide the initial deployment, but wouldn't ensure that the same amount of Pods is running at all times. For example, a script could deploy 5 Pods, but it wouldn't assure that in case of one of the Pods failure a new Pod is created.

Description

Thankfully, there is a resource type called ReplicaSet. As its name already suggests, ReplicaSet manages the number (or replicas) of deployed pods(13). This is quite helpful in cases when an application can be scaled by increasing the number of running instances. Additionally, ReplicaSet does not only deploy a specified number of Pods but also assures that the correct amount is always running.

3. A file containing commands necessary to provide certain functionality

Let's examine the following YAML representation of a ReplicaSet:

Listing 4.2: ReplicaSet snippet in YAML format

```
1  apiVersion: apps/v1
2  kind: ReplicaSet
3  metadata:
4    labels:
5      app: grafana
6    name: grafana-rs
7  spec:
8    replicas: 1
9    selector:
10     matchLabels:
11       app: grafana
12   template:
13     metadata:
14       labels:
15         app: grafana
16     spec:
17       containers:
18       - image: grafana/grafana:latest
19         imagePullPolicy: IfNotPresent
20         name: grafana
21         ports:
22         - containerPort: 3000
```

From the snippet 4.2, you can see that the metadata looks almost the same as the Pod snippet. This means that the created object (in this case a ReplicaSet) will have the name *grafana-rs* and will contain one label - *app: grafana*.

From the specification starting on line 7 we can see that the ReplicaSet should deploy exactly 1 replica. The information about the Pod it should create is present in the *.spec.template* field.

The ReplicaSet's specification also contains a selector detailing which Pods should the ReplicaSet acquire. When a Pod is acquired by a ReplicaSet, the ReplicaSet counts it as one of the replicas(13). As an example, if there is a Pod with the label *app: grafana* running before the ReplicaSet from the snippet 4.2 is created, the ReplicaSet will not

create another one, because the requested number of replicas (1) is already running.

When the snippet 4.2 is deployed, it is possible to execute the command `kubectl get all` which shows one ReplicaSet and one Pod running. To try the ReplicaSet behavior, command `kubectl delete pod -l app=grafana` will delete the Pod, which results in the immediate deployment of a new Pod.

4.3.3 ReplicationController

ReplicationController also assures that a specified number of replicas is running. According to the official Kubernetes documentation, it is a predecessor of ReplicaSet, as they both serve the same purpose. The main difference is the selector, which is used to associate the Pods with the given ReplicationController. While the ReplicaSet uses a set-based label selector enabling advanced label selection, ReplicationController does not support it and uses only the equality-based label selector. A user should favor the usage of a ReplicaSet over ReplicationController(14).

4.3.4 Deployment

Motivation

Even though a ReplicaSet is a powerful tool, it still lacks some functionality. For example, ReplicaSet cannot override the configuration of already running Pods. It is possible to change the Pod template specification in the ReplicaSet after deploying and creating the required replicas. Still, the already running Pods will remain with the old configuration, and only Pods created after the template change will contain the new one.

Description

To solve the problem mentioned in the *Motivation* and more, there is a Deployment. Let's look at the YAML representation of a Deployment first:

Listing 4.3: Deployment snippet in YAML format

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    labels:
5      app: grafana
6    name: grafana
7  spec:
8    replicas: 1
9    selector:
10     matchLabels:
11       app: grafana
12   template:
13     metadata:
14       labels:
15         app: grafana
16     spec:
17       containers:
18         - name: grafana
19           image: grafana/grafana:latest
20           imagePullPolicy: IfNotPresent
21           ports:
22             - containerPort: 3000
```

You can observe that the snippet 4.3 for Deployment looks almost identical to the snippet 4.2 for ReplicaSet. The only difference is the *.kind* field stating the resource type and the object's name in the *.metadata.name* field.

When the Deployment is deployed, it creates a ReplicaSet, which makes the desired Pods. Kubernetes online documentation states that the added value of the Deployment is the ability to update all the running Pods by changing the Deployment's *.spec.template* field. As an example, when we add a new environment variable[4] to the Deployment's Pod template, it automatically scales down the number of replicas for the current ReplicaSet and creates a new one with the updated configuration(15). This results in new Pods deployment with the updated values.

4. A new environment variable can be added using the command `kubectl set env deployment grafana MY_ENVVAR="My new envvar"`.

Suppose the update caused some issues with the Pod and it is necessary to return to the initial configuration. In that case, a user can execute a rollback with the command `kubectl rollout undo deployment grafana` (15).

Since Deployment has the most flexibility and offers the most features for Pod management, developers usually tend to create Deployments instead of ReplicaSets or Pods directly.

4.3.5 Service

Motivation

When Pods need to exchange data, some sort of communication has to be established. Each Pod inside a Kubernetes cluster has its IP address that could be used for communication(11). But as Deployments can destroy and create new Pods for various reasons, the IP addresses can differ for each new replica. This means that the tracking of current addresses could be complicated. Moreover, when there are multiple replicas of a Pod, it is probably ideal to balance the incoming requests load by spreading it to all the Pods.

Description

A service is an object grouping a set of Pods which can be reached with the Service's IP address(16). Let's take a look at the following snippet of a Service:

Listing 4.4: Service snippet in YAML format

```
apiVersion: v1
kind: Service
metadata:
  name: grafana
spec:
  ports:
    - port: 3010
      targetPort: 3000
  selector:
    app: grafana
  sessionAffinity: None
```

```
12    type: LoadBalancer
```

The snippet 4.4 specification starting on line 5 contains a selector defining the set of Pods targeted by this Service. The *.spec.ports* field specifies the mapping of the Service's ports to the Pods' ports. In this case, the Pods' port 3000 will be mapped to the Service's port 3010.

Service | Kubernetes [16] states that each Service has one of the following types:

- ClusterIP (default) - The IP address is cluster-internal only and can't be reached outside the cluster.
- NodePort - The Service is exposed on each Kubernetes node at a static port. The Service is accessible from outside the cluster.
- LoadBalancer - The Kubernetes hosting cloud provider's load balancer is used to expose the Service externally.
- ExternalName - A special type used for creating a Service for an external service.

For this snippet, the LoadBalancer type has been selected. This means that Kubernetes will create a load balancer supplied by the cloud provider hosting the cluster.

4.3.6 Ingress

Motivation

Connecting a frontend[5] and a backend[6] using an IP address provided by a Service might be enough. But when a user wants to access the frontend (user interface), it is common to use a domain name. For such functionality, Kubernetes provides an Ingress.

Description

In order to use Ingress functionality, it is necessary first to install an Ingress controller, such as *ingress-nginx*[7] (17).

5. A client-side service
6. a server-side service
7. More about the installation of NGINX Ingress Controller at `https://kubernetes.github.io/ingress-nginx/deploy/`.

Listing 4.5: Ingress snippet in YAML format

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: grafana-ingress
spec:
  rules:
    - host: my-grafana.xyz
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: grafana
                port:
                  number: 3010
```

The Ingress defined in snippet 4.5 expects already created Deployment and Service from previous sections. When the Ingress is deployed, the Kubernetes nodes will resolve the path *my-grafana.xyz* to the Grafana Service's IP address.

4.3.7 Namespace

Motivation

When there are multiple services deployed on a Kubernetes cluster, it might be a good idea to separate them into their own environments so it is easier to discover their resources. Such environments are called Namespaces.

Description

Besides separation of deployed services, *Share a Cluster with Namespaces* [18] says Namespaces also provide a way to define CPU and memory usage limits, enable the configuration of user access to different Namespaces, and more. When a Namespace is deleted, all of the resources belonging to the Namespace are also destroyed(18).

As an example, a new Namespace called *grafana* can be created using `kubectl create namespace grafana` command. To create new resources in this Namespace, `--namespace='grafana'` or `-n grafana` flag needs to be added to the used command. The resulting command would look like this: `kubectl create -f resource.yaml -n grafana`.

It is important to note that each resource type is either Namespace-scoped or not. Resources that are not Namespace-scoped are always common for the whole Kubernetes cluster. All resource types specified in previous sections are Namespace-scoped.

4.3.8 Persistent Volume Claim and Persistent Volume

Motivation

Pods are not persistent objects. They can be destroyed and created for multiple reasons. This means that the data stored in a container can be lost easily. For data that should always be preserved, such as database content, static website data files, etc., Kubernetes provides Persistent Volume (PV). To define a request for a PV, a Persistent Volume Claim (PVC) needs to be created(19).

Description

A Persistent Volume (PV) is storage with configured capacity that a Pod can use. When a Pod needs a Persistent Volume, it needs to specify a Persistent Volume Claim requesting the storage in its specification(19).

Either the PV can be created directly, or it can be created by a PVC if there is no suitable PV present. The creation of a PV using a PVC is known as dynamic provisioning(19).

Listing 4.6: Persistent Volume Claim snippet in YAML format

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc01
spec:
  accessModes:
```

```
7       - ReadWriteOnce
8     resources:
9       requests:
10        storage: 1Gi
```

The snippet 4.6 defines a Persistent Volume Claim with the name *pvc01*. Assuming that the Kubernetes cluster doesn't have any PV already created, the deployment of this snippet will make one. When the PVC is deleted, the PV is destroyed as well.

Besides the specification of the requested storage capacity in the *.spec.resources.requests* field, it also defines an access mode of the PVC. The access mode can be *ReadWriteOnce*, *ReadOnlyMany*, *ReadWriteMany*, or *ReadWriteOncePod*(19). Each of those access modes defines constraints on the writability and accessibility of the Persistent Volume.

As already mentioned, it is also possible to create a PV before the PVC is deployed.

Listing 4.7: Persistent Volume snippet in YAML format

```
1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4    name: pv01
5  spec:
6    accessModes:
7    - ReadWriteOnce
8    capacity:
9      storage: 1Gi
10   hostPath:
11     path: /data/pv01
12   persistentVolumeReclaimPolicy: Recycle
13   storageClassName: standard
14   volumeMode: Filesystem
```

The snippet 4.7 states the type of the PV as *hostPath*, which according to *Persistent Volumes* [19] might be used for local development on a single node cluster, but will not work on a multi-node setup. There are many

other types of Persistent Volume such as *nfs*[8], *awsElasticBlockStore*[9], *cephfs*[10], *local*[11], etc.(19)

Additionally, the snippet 4.7 also states the PV reclaim policy as *Recycle*[12]. The policy means that even if the PVC bound to this PV is deleted, the PV itself will remain(19).

Persistent Volume Claims are Namespace-scoped, while Persistent Volumes are not.

4.3.9 Node

As already mentioned, a Kubernetes cluster needs one or more machines to run on. According to *Nodes* [20] the machines can be either physical or virtual. For local development, one machine is enough, but for a production ready environment, running Kubernetes on multiple physical machines is preferred(20).

Nodes [20] states that nodes are the internal representation of each machine. Each Node can serve either as a Control plane (also referred to as a master node) or a node (also referred to as a worker node)(20). Control plane nodes are running services supporting the whole Kubernetes cluster, while worker nodes are intended for the deployed workload(10).

Each Node is tracking its health and the availability of its resources. Using command `kubectl describe node name`, where *name* is the name of the Node, it is possible to examine the usage of CPU, memory, ephemeral storage, etc. There is also information about the Node's condition, system info, Pods running on selected Node, and much more.

8. Network File System
9. AWS Elastic Block Store, this type can be used on AWS cloud provider
10. CephFS file system
11. local storage device, might replace *hostPath* on a multi-node cluster
12. Recycle reclaim policy is deprecated and dynamic provisioning should be favoured(19).

5 OpenShift

"Red Hat® OpenShift® is an enterprise-ready Kubernetes container platform with full-stack automated operations to manage hybrid cloud, multicloud, and edge deployments. Red Hat OpenShift is optimized to improve developer productivity and promote innovation." (21)

Red Hat is, according to *Stackalytics | Kubernetes community contribution during OpenStack Yoga release* [22], the third most significant contributor to the Kubernetes project after Google and independent developers. They have created their own Kubernetes distribution called OpenShift Container Platform. The platform brings many additional features and is tested to ensure enterprise-level quality.

5.1 Interaction tools

The most common tool used for communication with a Kubernetes cluster is *kubectl*. The same tool can be used for OpenShift Container Platform (OCP) since it is a distribution of Kubernetes. But as OCP is extending the features of Kubernetes, there is another tool called *oc*. In many cases, the usage of *oc* is identical to *kubectl*, but it also adds support for additional resource types, authentication, and other commands.

While *kubectl* needs a manual configuration of the *kubeconfig* file to define the cluster authentication details, *oc* provides an authentication command automatically generating the correct *kubeconfig* content.

5.2 Types of resources

This section specifies some of the enhancements OCP brings to Kubernetes. Similar to the Kubernetes chapter, this section doesn't provide all additional features and complete description but rather introduces various resource types essential for the scope of this thesis. This section is not trying to replace the official documentation of OpenShift, which is the ultimate source of information about this platform.

5.2.1 Project

While Kubernetes provides Namespaces to divide software running on a single cluster or provide separated environments to different users, OCP offers a resource type called Project. Namespaces are still present in OCP, but instead of being created directly by the user, they are created by a Project. Besides additional annotations that Projects have, they also enable functionality such as displaying (or access) to only a subset of all Projects to defined user groups(23). The command to create a new Project is `oc new-project my-project-name`.

As Projects only build on top of Namespaces and the creation of a Project automatically creates the underlying Namespace, the terms are often used interchangeably.

5.2.2 ImageStream

"An image stream and its associated tags provide an abstraction for referencing container images from within OpenShift Container Platform. The image stream and its tags allow you to see what images are available and ensure that you are using the specific image you need even if the image in the repository changes." (24)

In other words, ImageStream is a resource type representing an image.

Listing 5.1: ImageStream snippet in YAML format

```yaml
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: openjdk-11
spec:
  lookupPolicy:
    local: false
  tags:
    - name: latest
      from:
        kind: DockerImage
        name: registry.access.redhat.com/ubi8
            /openjdk-11
```

The snippet 5.1 contains a metadata section with the ImageStream's name. From the specification starting on line 5, you can see that the ImageStream has one tag named *latest*. This tag points to a docker image located in `registry.access.redhat.com` registry.

According to *Using image streams with Kubernetes resources* [25], ImageStream is an OpenShift Container Platform native resource. It works correctly with other OCP native resources, and it is possible to enable their functionality for Kubernetes native resources using the *spec.lookupPolicy.local* field. In the snippet 5.1 the field is set to false, which means that the ImageStream only works with other OCP native resources(25).

5.2.3 BuildConfig

While Kubernetes is expecting to deploy already built images from either its internal or public image registry, OpenShift Container Platform brings the ability to build new images directly on the platform.

The BuildConfig resource type serves as an object defining how and when an image should be built.

Listing 5.2: BuildConfig snippet in YAML format

```
1  apiVersion: build.openshift.io/v1
2  kind: BuildConfig
3  metadata:
4    name: sample-app
5  spec:
6    source:
7      type: Git
8      git:
9        uri: https://github.com/jsafarik/
           openshift-sample-app.git
10   output:
11     to:
12       kind: ImageStreamTag
13       name: sample-app:latest
14   strategy:
15     sourceStrategy:
16       from:
```

```
17          kind: ImageStreamTag
18          name: 'openjdk-11:latest'
19      type: Source
20    triggers:
21      - type: ConfigChange
22      - type: ImageChange
23      - type: "GitHub"
24        github:
25          secret: "my-webhook-secret"
```

From the specification part of snippet 5.2 you can see that the source code for the resulting image should be taken from a git repository. Other source types are *Dockerfile*[1] and *Binary*[2](26). The *.spec.strategy* field configures the base image used for the building process.

The image built from the BuildConfig in the snippet 5.2 is then pushed to the internal OCP registry and tagged as *latest* in an ImageStream called *sample-app*(27).

According to *Triggering and modifying builds* [28] there are 3 main types of build triggers:

- ConfigChange - indicates that the BuildConfig should be built on its creations.
- ImageChange - indicates that a new build should be triggered on change in the base image.
- Webhook - states that the trigger can be invoked using a webhook. In the example snippet 5.2 a GitHub webhook trigger is used.

Thanks to BuildConfig, it is possible to set up a CI/CD workflow enabling rapid build of a new image on each new commit.

5.2.4 Build

When a BuildConfig is created and triggered, it creates a new Build object, which takes care of the rest of the build process.

1. Dockerfile source type should contain the commands to build the image.
2. BuildConfig with binary source type can be triggered only from CLI as it streams specified data to the cluster.

"Build objects share common characteristics including inputs for a build, the requirement to complete a build process, logging the build process, publishing resources from successful builds, and publishing the final status of the build. Builds take advantage of resource restrictions, specifying limitations on resources such as CPU usage, memory usage, and build or pod execution time." (29)

While BuildConfig contains the overall description of the build process, the Build resource includes specific information such as a commit ID of the source code, the base image digest, etc.

Even though the Build contains a specific set of information to build the image, the actual build is done inside a build Pod automatically created by the Build object.

5.2.5 DeploymentConfig

The Kubernetes chapter describes a resource type called Deployment, which uses ReplicaSets to maintain the correct number of Pods deployed. According to *Understanding Deployment and DeploymentConfig objects* [30] OpenShift brings additional resource called DeploymentConfig. It uses ReplicationController to assure the correct number of replicas. While they both serve a similar purpose, each has its advantages and disadvantages. For example, unlike Deployment, DeploymentConfig supports an automatic rollback to the last successful instance of ReplicationController, execution of custom hooks, custom triggers, etc. On the other hand, Deployment can be, for example, paused in the middle of the deployment process. This is not supported by DeploymentConfig(30).

5.3 Common problems with OpenShift clusters in QE environment

The most common issues with OCP clusters encountered in the QE environment can be categorized into issues caused by infrastructure instability and problems caused by human error.

5.3.1 Human error

Many test suites[3] used for product testing are directly interacting with the OCP cluster. Besides the actual tests, some test suites contain functionality to deploy and destroy the tested product. For this, a user with cluster administrator rights is often used.

The problem can usually occur during the product destruction phase. While a correct implementation of the destruction will not cause any issues, test suites are often changed to support the tested software, so an introduction of a new bug is possible.

Examples of the issues a test suite can cause are:

- Destruction of a resource outside of the specified Project
- Destruction of a non-Namespace-scoped objects
- Misconfiguration of resources necessary for correct OCP behavior

Any of these problems might cause a degradation of the OCP cluster or even its complete inaccessibility.

Similar issues may arise from the user's direct manipulation with the cluster as an admin user. Quality engineers often need admin access to develop new tests or verify application features, but even a single command deleting an essential component might cause big problems.

5.3.2 Infrastructure instability

Accessibility

One of the most common issues is not being able to access the OpenShift Container Platform cluster. This problem can be further divided into not being able to access the OCP cluster's console[4], not being able to contact the cluster's API or both.

The OCP cluster's console is managed by the OCP itself. The console inaccessibility might be caused by the cluster or by a networking

3. A software developed to test a product
4. Console, also referred to as dashboard, is the user interface of the OpenShift cluster

issue of the cloud provider hosting the OCP. Even though a test suite might not need access to the console, it still indicates that the cluster is not working correctly.

When the API is not working, it is safe to say that the cluster is not suitable for testing. This can have many causes, such as a networking issue of the cloud provider hosting the OCP, degraded state of the machines running the cluster's nodes, etc.

Degraded Nodes

A cluster usually has multiple Master and Worker nodes. An infrastructure instability might cause an erroneous state of a machine hosting the node, which causes a degradation of the OCP. This problem might occur even if the machine is running, but some of the cluster services are degraded.

A node can also get into a degraded state by simply using the OCP. For example, an overloaded cluster might run out of disk space or other resources.

When a cluster has a lower number of nodes than expected, it doesn't necessarily mean that it is unsuitable for testing. In a scenario where a cluster has three working Master nodes, two working, and one degraded Worker node, it should still be usable. On the other hand, two or more degraded nodes decrease the cluster's correct behavior probability.

Connectivity to outside services

It is often desirable to use other online services such as GitHub from an OCP cluster. It can cause many problems when this and similar services are not accessible.

There was a problem with connectivity from a Pod to `github.com`[5]. When any resource tried to use this service, it resulted in a request timeout. This may cause many problems, such as not being able to build a new image from a source code from GitHub, using configuration files stored on the service, etc.

5. The issue is documented at `https://bugzilla.redhat.com/show_bug.cgi?id=1991067`

There is a workaround for this issue. A Google DNS IP address can be added to the cluster for `github.com` and possibly other websites. Even though a workaround exists and the core problem of the issue should be fixed, this problem has caused many issues during test execution.

Persistent Volumes not working

Applications that need to persist its data request a Persistent Volume (PV). If the PV needs to be created first, a problem might occur when the cloud provider is either unable to create a volume or takes too long. In this case, the PV is never created, and the application requesting it will not work correctly.

In case that the cluster already has Persistent Volume created, another problem might occur in case that the cloud provider is not able to attach it to a machine. This results in the same issue as the previous problem, and the application will not work as supposed.

5.4 Current monitoring solutions

For OpenShift Container Platform there is a lot of monitoring solutions and systems providing a look at the OCP cluster status, usage, alerting, and more.

5.4.1 OpenShift Monitoring Stack

OpenShift itself has a robust monitoring stack. According to *Understanding the monitoring stack* [31], it provides monitoring of the core platform components, graphs of cluster metrics in the dashboard, and alerting of the cluster administrators about caught issues. Additionally, it is also possible to enable monitoring of user-defined projects(31). The main disadvantage for QE teams running multiple clusters is that the metrics are provided in each cluster separately, and the assessment of the status of all clusters is a time-consuming process. Another disadvantage is that the default monitoring stack doesn't discover all the problems described in the previous section. For example, the issues with Persistent Volume is discovered only after an application tries to use one.

5.4.2 VictoriaMetrics

Of course, there are plenty of third-party solutions, such as VictoriaMetrics. *VictoriaMetrics* [32] explains it is an open-source time-series database and monitoring solution. The advantages of VictoriaMetrics include much better system utilization, such as seven times less RAM usage than Prometheus, metrics' relabeling, long-term storage, data exposure using many protocols, and more(32).

Additionally, it is also possible to set a single VictoriaMetrics instance to scrape data from all monitored OpenShift instances to gather the data to a single point(33). While it brings the solution to assess the status of all the clusters in a single place, the metrics provided are only the ones supplied by OpenShift and don't necessarily need to discover all problems.

5.4.3 DataDog

Another third-party solution is DataDog. DataDog brings many advantages and additional features, such as connection to the cloud provider environment, monitoring multiple clusters using deployable agents, processing of other APIs exposing their metrics data, etc(34).

While DataDog provides many features, there are disadvantages as well. One of the main disadvantages is that it is a non open-source software provided as a service(34). Another disadvantage is pricing. Even though there is a free tier, it is very limited(35).

5.4.4 Other open-source projects

There are also open-source projects developed by individuals or teams found on various Git services. One example of such a project could be *Kubernetes Operational View*[6]. Although *Kubernetes Operational View* [36] states that it is not a monitoring solution, it provides an overview of multiple cluster resources. This project focuses on system resources such as nodes, CPU, RAM, and Pods(36).

While this solution provides data closest to the sought solution, it still doesn't do checks of the cluster functionality. This means that a

6. Kubernetes Operational View can be found at `https://codeberg.org/hjacobs/kube-ops-view`

problem with the deployment will probably be discovered after test execution, and this tool will only visualize and inform about the status of the cluster.

6 OpenStack

"Openstack controls large pools of compute, storage, and networking resources, all managed through APIs or a dashboard.

Beyond standard infrastructure-as-a-service functionality, additional components provide orchestration, fault management and service management amongst other services to ensure high availability of user applications." (37)

6.1 Infrastructure as a Service

Infrastructure as a Service (IaaS) enables users to provision and manage compute instances (servers), storage, network, and more(38). Thanks to this, it is possible to create infrastructure resources necessary for platforms such as Kubernetes or OpenShift.

Such a service enables the user to create and manage only the required resources. This lowers users' expenses while allowing easy tracking of all provisioned resources.

There are other models, such as Platform as a Service (PaaS) or Software as a Service (SaaS)(39). An example of PaaS is OpenShift Container Platform. An example of SaaS could be access to an application deployed on OCP.

6.1.1 Types of resources

Similar to chapters dedicated to Kubernetes technologies, this section also does not replace the official OpenStack documentation, but rather provides the motivation behind some of the essential OpenStack resources.

Servers

The most fundamental component a user can provision is a server, also referred to as an instance. A server is a virtual machine with predefined CPU, Ram, and Disk storage. Each server also specifies an image used for the virtual machine. Servers are backed by the OpenStack project called Nova(40).

This is the basic building block for any deployed service. When a Kubernetes cluster is deployed, each Node will require one server to run on.

Volumes

Volumes represent a persistent storage. Each volume has a specified amount of disk storage and can be mounted to any server. Using this, it is possible to store data independently of any instance, and in case of a server failure or deletion, the data are preserved. Volumes are backed by the Cinder OpenStack Block Storage service(41).

A Kubernetes or OpenShift cluster can utilize volumes to create Persistent Volume.

Networking

OpenStack Networking [42] explains that the OpenStack contains a complex networking solution backed by the OpenStack project called Neutron. The project provides network connectivity as a service. Thanks to the service, a user can create and manage networks, subnets, ports, security groups, floating IPs, load balancers, and more(42).

6.2 Common problems

6.2.1 Resource cannot be deleted

One of the most common problems is related to OpenShift Container Platform, particularly to OCP destruction. When a cluster is no longer needed and a destruction process is triggered, all OpenStack resources related to the cluster have to be deleted. Unfortunately, the cluster might leave some of the resources in a status preventing their destruction. For example, a volume in state *reserved* cannot be deleted and without proper rights, the state cannot be changed.

This issue is causing the OpenStack to contain no longer necessary resources. Each OpenStack tenant[1] has its own limits for each resource type and stuck resources might exhaust the quota.

1. A tenant is an isolated environment in an OpenStack instance

6.2.2 Slowness

Another problem might be a slowness of some of the OpenStack components. For example there might be an issue causing a slower server provisioning. Even though this doesn't have to seem like a big problem, it can actually cause major issues. A lot of automations, such as OCP deployment, rely on the creation of the resources in a certain time interval. In case when the interval is exceeded, the deployment will fail.

6.3 Current monitoring solutions

In the scope of this thesis, the approach to OpenStack monitoring is different than to OpenShift Container Platform monitoring. The main difference is that there are multiple OCP clusters that should be monitored simultaneously, while only one instance of OpenStack.

The most important and possibly the only necessary monitoring solution that needs to be mentioned is the OpenStack Telemetry project. The project consists of multiple other sub-projects, such as Ceilometer, Aodh, or Gnocchi.

6.3.1 OpenStack telemetry

Ceilometer

According to *System architecture — Ceilometer 17.0.0.0rc2.dev2 documentation* [43], Ceilometer's responsibility is to collect and normalize data used for monitoring and alerting. It uses an agent-based architecture to gather the information from other OpenStack components. It also integrates with various databases, such as Gnocchi, which is used to store measurements(43).

Aodh

Aodh is a project responsible for alerting (or alarming) based on the monitoring data(44). *Alarms — aodh 13.0.0.0rc2.dev2 documentation* [45] states that the project provides a mechanism to trigger actions on user-defined rules automatically. Each alarm can be in one of three states: *ok*, *alarm*, or *insufficient data*. Thanks to this project, an administrator

can be aware of the current status of the infrastructure, and custom actions can be performed for each alarm(45).

Gnocchi

Gnocchi is an open-source time-series database used in the OpenStack telemetry stack as the storage of measurements(43). It glues the Ceilometer and Aodh projects together. The Ceilometer pushes the data to Gnocchi and the Aodh project can use the data from Gnocchi to evaluate the current status(45). Unlike Ceilometer and Aodh, the Gnocchi project is not managed by the OpenStack telemetry team(44).

7 Proposed solution

7.1 Requirements

One of the early stages of the system design and development was gathering and analyzing the functional and non-functional requirements. Functional requirements describe the services the system provides, the system's outcomes on given inputs, and the overall system behavior. The non-functional requirements specify the system's properties and limitations. In other words, functional requirements ask what the functionality is, while non-functional ask how it is done.

7.1.1 Functional requirements

- The user can access monitoring results in form of graphs and tables.
- The user can specify a set of clusters.
- The user can filter the set of clusters intended for the monitoring.
- The user can access a list of all currently working clusters.

The requirements mention a user, which can be either a person, or another service (for example a CI/CD tool) using the system.

7.1.2 Non-functional requirements

- The system iterates over one or more kubeconfig files to discover all OpenShift clusters. Environment variables or properties file can further filter the clusters.
- The system periodically checks all tracked OpenShift clusters. The periodicity of the checks should be defined for each check separately.
- The system exposes monitoring data in a format acceptable for Prometheus.

- The system contains a REST API with endpoints providing a view of currently running clusters.
- The system encapsulates each monitoring check as an independent class.
- The system can run in a containerized environment.

7.2 Development stack

This section provides a summary of the selected tools, technologies, and libraries used for the development of the proposed solution.

7.2.1 Java

As the base of the whole project, I have selected Java as the programming language implementing all the functionality necessary to gather the monitoring data. Java belongs to one of the most popular programming languages, ranking second in the Popularity of Programming Language (PYPL) website (46).

Some of the features contributing to the success of Java are the ability to run the code on many different systems thanks to the Java Virtual Machine (JVM), easy to learn syntax, and object oriented approach.

There are many versions of Java supported at the moment of writing this thesis. Java 11 is a Long Term Support (LTS) release and has been selected as the version used to implement the applications.

7.2.2 Quarkus

"Quarkus is a full-stack, Kubernetes-native Java framework made for Java virtual machines (JVMs) and native compilation, optimizing Java specifically for containers and enabling it to become an effective platform for serverless, cloud, and Kubernetes environments." (47)

Quarkus is a robust open-source Java framework for the building of enterprise applications. The strength of Quarkus is in its wide set of extensions and the before mentioned support for containerization.

Extensions

Quarkus is providing most functionality and new features in the form of extensions. At the moment of writing this thesis, the `code.quarkus.com` website lists more than 400 extensions across 15 categories.

Context and Dependency Injection (CDI)

Similar to Spring Boot, Quarkus offers a dependency injection mechanism. It is based on *Contexts and Dependency Injection for Java 2.0*[1] even though it is not implementing the whole specification, but only a subset of features(48).

Native builds

Quarkus also provides support for native builds using GraalVM. According to the GraalVM website, it is a "High-performance runtime with new compiler optimizations to accelerate Java application performance and lower infrastructure costs on premises and in the cloud" (49).

7.2.3 Fabric8 Kubernetes client

Fabric8 Kubernetes client provides a rich Domain specific language (DSL) for making REST API requests to any OpenShift or Kubernetes clusters. The client facilitates the system's communication with each OCP cluster to deploy the sample application and gather the monitoring data.

The client is widely used for several projects, such as Apache Camel or Spotify's Styx (50).

7.2.4 Camel

"Camel is an Open Source integration framework that empowers you to quickly and easily integrate various systems consuming or producing data." (51)

1. Java specification request for this feature is accessible at `https://docs.jboss.org/cdi/spec/2.0/cdi-spec.html`

Apache Camel enables its users to create and execute integration routes based on the Enterprise Integration Patterns. An example of such route can be:

```
from("timer:myTimer?period=1000&repeatCount=3").
   log("Hello Camel");
```

The example will produce the log *Hello Camel* 3 times with 1 second period. This is one of the most simple routes possible as it uses only two components - Timer and Log. Overall, there are over 300 camel components.

7.2.5 Quartz

Quartz Job Scheduler is a Java library providing support for scheduling and executing jobs with user implemented functionality. The library takes care of threads management and provides a simple API for defining new jobs and triggers.

A job can be any class implementing the *Job* interface provided by the Quartz library. To register a job for execution, a *JobDetail* instance has to be created with the information about the implemented job class. Finally, a *Trigger* instance facilitates the specification of when a job should run, how many times it should run, and other details about the job's execution.

7.2.6 Prometheus

As stated in the Health monitoring section, Prometheus is a monitoring system used to collect and store metrics data. The most crucial step for integrating Prometheus with another application is to specify where to find the applications metrics data. The application should have a dedicated REST API endpoint providing all the data.

Prometheus can scrape metrics from multiple systems, providing the ability to store metrics data of many applications on a single Prometheus instance.

Prometheus also provides a user interface, which a user can use to query the gathered data using PromQL. PromQL is a query language used to select and transform data to create custom graphs and tables (52).

7.2.7 Grafana

Even though Prometheus also provides metrics data visualization, its capabilities are limited in this area and usually serve as a debugging and data checking tool.

Grafana is open source analytics and visualization platform (53). It provides robust support for creating dashboards showing the monitored data in a preferred format. It supports multiple graphs and charts such as line, bar, pie, histogram, heatmap, and many more. The user can create not only graphs but also tables, lists, and much more. It is also possible to install plugins extending the system's features, such as integration with TeamViewer or GitHub.

7.2.8 OpenShift

Besides the objective of this thesis to monitor multiple OpenShift Container Platform clusters, the monitoring application itself should be containerized and possible to deploy on OCP. This enables easy deployment and management of the application with all of its dependencies and components.

7.3 Architecture

The figure 7.1 describes a high-level overview of the proposed architecture of the monitoring system. The OCP cluster on the left is the environment hosting the application gathering the metrics. The application is called *Multi OCP Monitoring*, and it is deployed together with Prometheus and Grafana instances. On the right, there are the OCP clusters that are being monitored. Even though the figure shows only 3 clusters, the application should monitor any reasonable number of clusters. Each monitored cluster hosts periodically deployed sample application used for the verification of the clusters behavior.

7.3.1 Sample application

The sample application is designed specifically to use as many different resource types of OCP as possible. In case that one of the resources fails, the sample application will not work correctly. Thanks to this

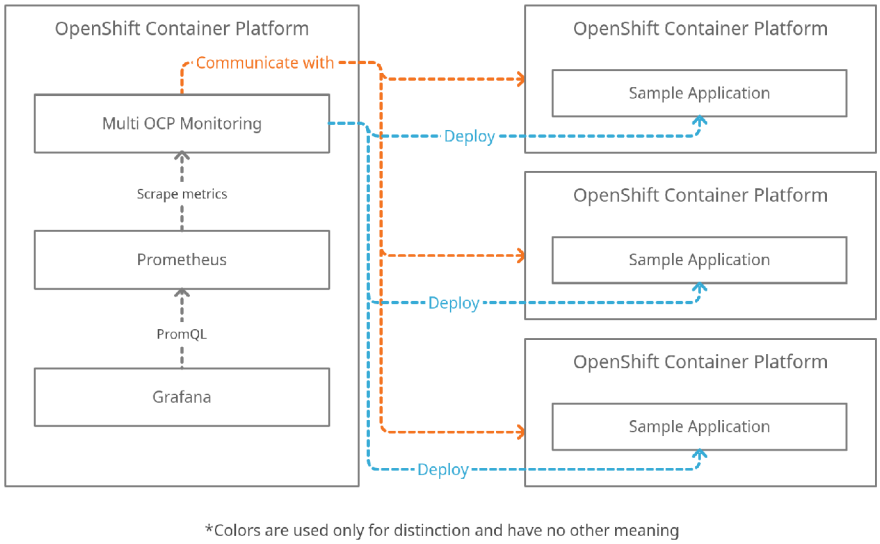

Figure 7.1: The architecture of the proposed solution

design, verifying the OCP's correct behavior can be done relatively quickly.

The problem this approach solves is the ability to detect problems with the OCP, its underlying cloud provider, or a third party services before the environment is used for testing.

Specifically, the sample app uses Persistent Volume as a storage for a database, without which the application will not be able to persist its data. If there is a problem with this resource, caused either by misbehaving OCP or instability of the cloud provider, the usage of the application will quickly reveal it.

Another problem discovered early in the deployment is the connectivity to GitHub. The source code of the application is located on this service, and since the application needs to be built by the OpenShift Container Platform, the inability to reach GitHub will fail the whole deployment process.

More about the design and deployment of the sample application can be found in the section 8.1.

7.3.2 Multi OCP Monitoring

The Multi OCP Monitoring application is responsible for gathering the metrics data and exposing it to Prometheus and users. The application maintains a list of monitored clusters and uses their API to verify their functionality.

One of the procedures the application repeatedly executes is the deployment of the sample application. Thanks to this, it can periodically determine the status of the clusters.

Other attributes the software checks are the API and console accessibility and the number of nodes in the *ready* state.

Due to the application's architecture, implementing new checks is as easy as creating a new class with a single method for the execution. More about the design and deployment of this application can be found in the section 8.2.

8 Implementation

This chapter builds on the proposed solution and architecture mentioned in the section 7.3. Each application has its section with a description of its design and deployment.

8.1 Sample application

The sample application functionality is a simple TODO list. It is a Quarkus application with 5 REST API endpoints backed by Apache Camel routes. The goal of the application is to use as many fundamental resource types of OCP as possible. Thanks to the deployment process, it is possible to check that the cluster behaves correctly.

The source code of the application is submitted with the diploma thesis and is also available on GitHub[1].

The five endpoints are:

- */add* is a *POST* request method endpoint expecting a plain text request body with the task contents.
- */get* is a *GET* request method endpoint returning all saved tasks.
- */get/{id}* is similar to the previous endpoint but returns only the task associated with the *id* provided as a path parameter.
- */delete/{id}* is a *DELETE* request method endpoint that deletes the task with the associated *id*.
- */update/{id}* is a *PUT* request method endpoint used to change already saved tasks with the given *id*. The endpoint expects a request body in plain text format.

8.1.1 Design

The figure 8.1 shows a class diagram of the sample application. As you can see, the implementation part is not particularly complex. There are only two classes - *TaskRoutes* and *Task*.

1. Located at `https://github.com/jsafarik/openshift-sample-app/tree/diploma-thesis`

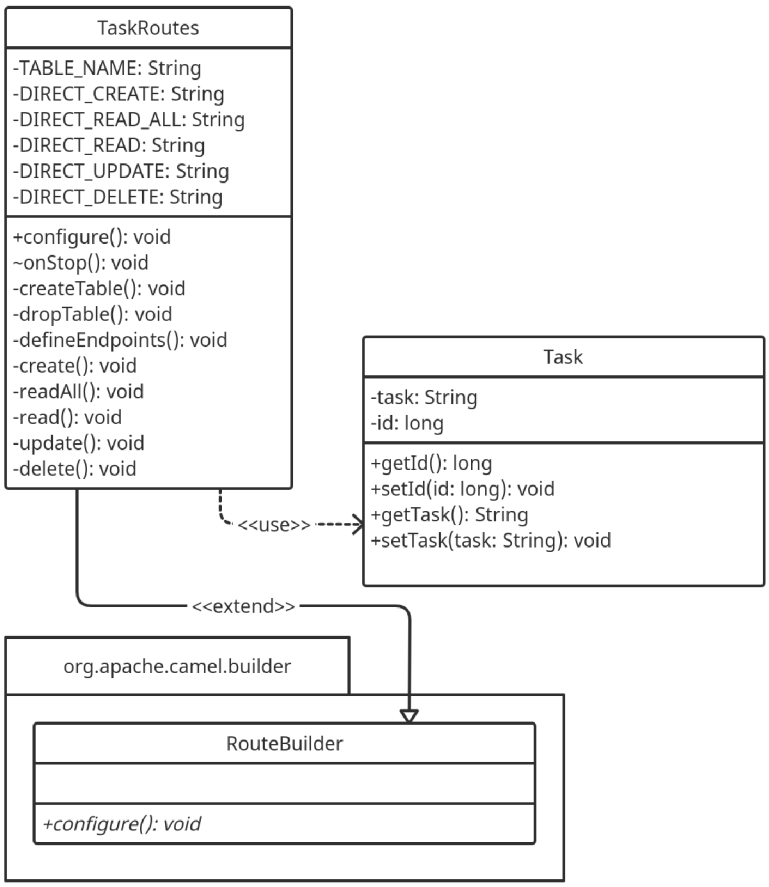

Figure 8.1: Class diagram for the sample application

The *TaskRoutes* class is extending the abstract class *RouteBuilder* provided by Apache Camel. It overrides a single method called *configure*, which defines all the REST API endpoints and their associated camel routes. Each route is separated into its function for better readability. Three additional methods are worth mentioning: *onStop*, *createTable*, and *dropTable*. The *onStop* method is invoked only on the shutdown of the whole application and calls the *dropTable* method to delete the table containing all the tasks from the database. The *createTable* method is

invoked at the start of the application and causes the initialization of the tasks table.

The second class, *Task*, is only a data class representing a task in the database.

8.1.2 Deployment

The more complex part of the sample application is its deployment process. The source code contains two files in YAML format defining the resources to create to start the application on OCP cluster.

The first file is called *postgres.yaml* and contains a list of resources necessary to deploy a PostgreSQL database. Those resources are:

- DeploymentConfig defining parameters for the deployed container, such as environment variables, Persistent Volume Claim's name, triggers for deployment and more. The DeploymentConfig uses *postgresql* ImageStream provided by OCP.
- Persistent Volume Claim seeking a Persistent Volume with 1 Gibibyte[2] of space.
- Service used for the database. The sample application uses this Service to connect to the database.

The second file, *sampleApp.yaml*, is an OCP template for the sample application. While a list can be used to deploy the resources directly, a template needs to be processed into the list first. The processing needs to be done because the template can contain parameters tweaking the final resources. The generated resources are:

- ImageStream for an *openjdk-11* image used as a base image for the sample application build.
- ImageStream for the actual sample application.
- BuildConfig for the sample application stating the URL of the source code, the ImageStream for the base image, and the output ImageStream where the final image is pushed.

2. 1 gibibyte is equal to 2^{30} bytes.

- DeploymentConfig, which specifies the sample application pod's information, such as the environment variables and the events leading to a new deployment trigger.
- Service targeting the sample application's pod.
- Route targeting the sample application's Service.

8.2 Multi OCP Monitoring

The monitoring application is called *Multi OCP Monitoring*. It delivers the checks necessary to establish the state of all the OCP clusters. The gathered information is exposed to Prometheus, which stores the data. Grafana is used to display the data in a human-friendly format.

The source code of the application is submitted with the diploma thesis and is also available on GitHub[3].

The application also provides a REST API for users to request the currently working clusters or a subset of them. The 3 REST API endpoints are:

- */q/metrics* provides metrics data in a format Prometheus can understand.
- */clusters* lists API URLs of all the clusters that are working. A working cluster is considered when it is accessible and can deploy an application.
- */clusters/{filter}* behaves similarly to the previous endpoint, with the difference that it returns only those clusters that contain the *{filter}* path parameter.

8.2.1 Design

The class diagram for this application pictured in the figure 8.2 is much more complex than the diagram for the sample application.

The *Configuration* is a singleton providing access to the application's parameters. The currently supported parameters are used to set the

3. Located at `https://github.com/jsafarik/multi-ocp-monitoring/tree/diploma-thesis`

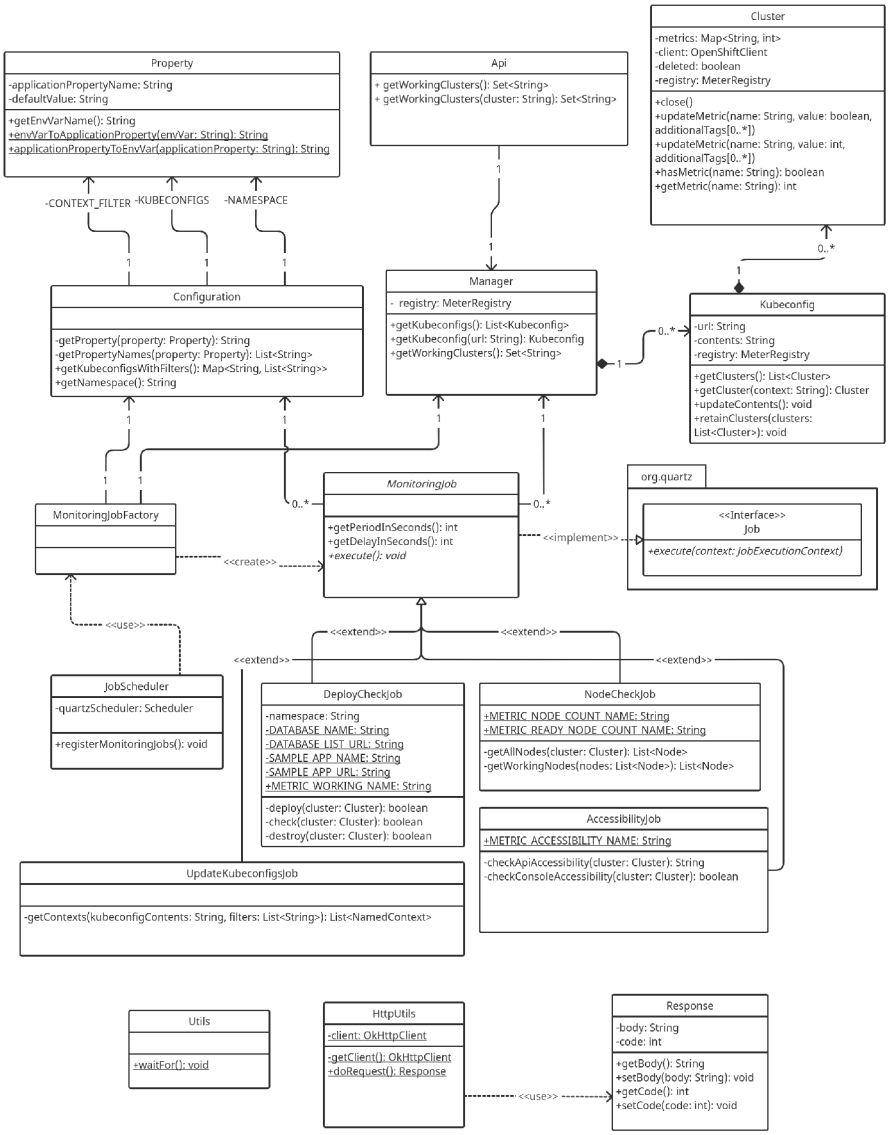

Figure 8.2: Class diagram for the Multi OCP Monitoring application

Project name dedicated for the sample application, a list of kubeconfig files, and cluster filters. The *Property* class represents the parameters.

The *Manager* class is also a singleton managing a list of *Kubeconfig* objects, and the *Kubeconfig* class is used to manage a list of *Cluster* objects. This hierarchy represents the OCP clusters that are monitored by the application.

The *Api* class exposes REST API endpoints to request a set (or subset) of currently working clusters.

The *MonitoringJob*, together with its four classes extending the abstract class, provides the logic gathering the monitoring data. The four classes are:

- *DeployCheckJob* is responsible for deploying the sample application to each of the tracked OCP clusters. After the deployment, the job verifies the sample application's correct behavior and destroys it.

- *NodeCheckJob* gathers the OCP cluster's nodes. After that, it saves the number of all nodes and the number of nodes in the *ready* state.

- *AccessibilityJob* uses the cluster's API to retrieve the console URL. If this URL is obtained correctly, the API is accessible. The console URL is used to send a simple HTTP request to verify that the console is accessible as well.

- *UpdateKubeconfigsJob* has a different purpose than the other jobs. This job periodically updates the tracked list of kubeconfigs and clusters but doesn't communicate with any OCPs.

The *MonitoringJob*s are scheduled for execution by the *JobScheduler* class. The *JobScheduler* uses the scheduler provided by the Quartz library to register all of the jobs. The class also provides an instance of *MonitoringJobFactory* to the scheduler. The *MonitoringJobFactory* class enables the job's objects modification before they are executed.

Finally, there are two util classes, *Utils* and *HttpUtils*. These classes are intended for static methods that the whole application can use.

8.2.2 Deployment

Since this application is a Quarkus project, it comes with the default *Dockerfile*s with instructions on building the final image. This makes it very easy to create the image, which can be used locally using Docker or deployed on OpenShift Container Platform.

Only two commands are necessary to build the image:

```
./mvnw package
docker build -f src/main/docker/Dockerfile.jvm -
   t quarkus/monitoring-jvm .
```

After the build is done, the image can be run locally using command:

```
docker run -i --rm -p 8080:8080 quarkus/
   monitoring-jvm
```

To deploy the application on OpenShift, the source code hosts 2 YAML files with instructions on how to use them.

Thanks to the application's periodic checking of each cluster, it is possible to assess the status of the QE team's infrastructure. The team can react accordingly to the reported health status and save valuable time when an OCP degradation is detected. Additionally, thanks to the application's API it is possible to integrate it into already existing automation. The integration can enable automatic destruction and deployment of clusters, suspension of tests execution, etc.

9 Conclusion

This thesis aimed to analyze current monitoring tools and propose a new solution with regard to the outcome of the analysis. The target of the monitoring is OpenShift Container Platform. I have designed and implemented a new system using various technologies revolving mainly around the Java programming language based on the proposed solution.

The implemented system solves the problem of OCP issue detection before product testing starts and provides an overview of the infrastructure status for the whole QE team.

9.1 Future work

Even though the applications are working and support the required functionality, possible enhancements can be done. New features for Multi OCP Monitoring could include:

- Add new jobs checking other components of OpenShift Container Platform.
- Add richer configuration support.
- Provide standalone and single time execution to determine the current state of infrastructure. The application could run each job only once in this mode and output a comprehensive report.
- Implement parallel cluster checking to increase the system's performance. Currently, the jobs can run parallel, but each job checks the clusters serially.
- Add the possibility to disable jobs. If the user doesn't want to execute all jobs, there should be a possibility to turn them off.

The sample application could include more deployment options, such as an operator driving the deployment or a new list and template to support deployment on the Kubernetes cluster.

Of course, the systems' architecture and the approach to monitoring can change with the continuous development of the applications.

Bibliography

1. LIGUS, Slawek. *Effective Monitoring and Alerting*. First Edition. Sebastopol (California): O'Reilly Media, Inc., 2012. ISBN 9781449333522.
2. *Overview | Prometheus* [online]. 2021 [visited on 2021-10-19]. Available from: https://prometheus.io/docs/introduction/overview/.
3. *Overview — Graphite 1.2.0 documentation* [online]. Graphite Project, 2012 [visited on 2021-12-07]. Available from: https://graphite.readthedocs.io/en/latest/overview.html.
4. *Feeding In Your Data — Graphite 1.2.0 documentation* [online]. Graphite Project, 2018 [visited on 2021-12-07]. Available from: https://graphite.readthedocs.io/en/latest/feeding-carbon.html.
5. *Graphite* [online]. Graphite Project, 2017 [visited on 2021-12-07]. Available from: https://graphiteapp.org/.
6. *OpenTSDB - A Distributed, Scalable Monitoring System* [online]. The OpenTSDB Authors [visited on 2021-12-07]. Available from: http://opentsdb.net/.
7. *Use containers to Build, Share and Run your applications* [online]. Palo Alto, [2021] [visited on 2021-10-09]. Available from: https://www.docker.com/resources/what-container.
8. MCCARTY, Scott. *A Practical Introduction to Container Terminology* [online]. Raleigh, 2018 [visited on 2021-10-10]. Available from: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction.
9. *FAQ - Open Container Initiative* [online]. San Francisco, c2020 [visited on 2021-10-10]. Available from: https://opencontainers.org/faq/.
10. *Kubernetes Components* [online]. 2021 [visited on 2021-10-26]. Available from: https://kubernetes.io/docs/concepts/overview/components/.

11. *Pods | Kubernetes* [online]. 2021 [visited on 2021-10-21]. Available from: https://kubernetes.io/docs/concepts/workloads/pods/.

12. *Creating Highly Available clusters with kubeadm | Kubernetes* [online]. San Francisco: Cloud Native Computing Foundation, 2021 [visited on 2021-12-08]. Available from: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/.

13. *ReplicaSet | Kubernetes* [online]. San Francisco: Cloud Native Computing Foundation, 2021 [visited on 2021-12-08]. Available from: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

14. *ReplicationController | Kubernetes* [online]. San Francisco: Cloud Native Computing Foundation, 2021 [visited on 2021-12-08]. Available from: https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/.

15. *Deployments | Kubernetes* [online]. San Francisco: Cloud Native Computing Foundation, 2021 [visited on 2021-12-08]. Available from: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

16. *Service | Kubernetes* [online]. San Francisco: Cloud Native Computing Foundation, 2021 [visited on 2021-12-08]. Available from: https://kubernetes.io/docs/concepts/services-networking/service/.

17. *Ingress* [online]. San Francisco: Cloud Native Computing Foundation, 2021 [visited on 2021-12-08]. Available from: https://kubernetes.io/docs/concepts/services-networking/ingress/.

18. *Share a Cluster with Namespaces* [online]. San Francisco: Cloud Native Computing Foundation, 2021 [visited on 2021-12-08]. Available from: https://kubernetes.io/docs/tasks/administer-cluster/namespaces/.

19. *Persistent Volumes* [online]. San Francisco: Cloud Native Computing Foundation, 2021 [visited on 2021-12-08]. Available from: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

20. *Nodes* [online]. San Francisco: Cloud Native Computing Foundation, 2021 [visited on 2021-12-08]. Available from: https://kubernetes.io/docs/concepts/architecture/nodes/.

21. *Red Hat OpenShift makes container orchestration easier* [online]. Raleigh [visited on 2021-10-31]. Available from: https://www.redhat.com/en/technologies/cloud-computing/openshift.

22. *Stackalytics | Kubernetes community contribution during OpenStack Yoga release* [online]. Stackalytics community members, 2021 [visited on 2021-12-09]. Available from: https://www.stackalytics.io/cncf?project_type=kubernetes.

23. *Projects and Users - Core Concepts | Architecture | OpenShift Container Platform 3.11* [online]. Raleigh, 2021 [visited on 2021-10-31]. Available from: https://docs.openshift.com/container-platform/3.11/architecture/core_concepts/projects_and_users.html.

24. *Understanding containers, images, and image streams* [online]. Raleigh: Red Hat, Inc., 2021 [visited on 2021-12-09]. Available from: https://docs.openshift.com/container-platform/4.9/openshift_images/images-understand.html.

25. *Using image streams with Kubernetes resources* [online]. Raleigh: Red Hat, Inc., 2021 [visited on 2021-12-09]. Available from: https://docs.openshift.com/container-platform/4.9/openshift_images/using-imagestreams-with-kube-resources.html.

26. *Creating build inputs* [online]. Raleigh: Red Hat, Inc., 2021 [visited on 2021-12-09]. Available from: https://docs.openshift.com/container-platform/4.9/cicd/builds/creating-build-inputs.html%5C#builds-binary-source_creating-build-inputs.

27. *Managing build output* [online]. Raleigh: Red Hat, Inc., 2021 [visited on 2021-12-09]. Available from: https://docs.openshift.com/container-platform/4.9/cicd/builds/managing-build-output.html.

28. *Triggering and modifying builds* [online]. Raleigh: Red Hat, Inc., 2021 [visited on 2021-12-09]. Available from: https://docs.openshift.com/container-platform/4.9/cicd/builds/triggering-builds-build-hooks.html.

29. *Understanding image builds* [online]. Raleigh: Red Hat, Inc., 2021 [visited on 2021-12-09]. Available from: https://docs.openshift.com/container-platform/4.9/cicd/builds/understanding-image-builds.html.

30. *Understanding Deployment and DeploymentConfig objects* [online]. Raleigh: Red Hat, Inc., 2021 [visited on 2021-12-09]. Available from: https://docs.openshift.com/container-platform/4.9/applications/deployments/what-deployments-are.html.

31. *Understanding the monitoring stack* [online]. Raleigh: Red Hat, Inc., 2021 [visited on 2021-12-09]. Available from: https://docs.openshift.com/container-platform/4.9/monitoring/understanding-the-monitoring-stack.html.

32. *VictoriaMetrics* [online]. San Francisco: VictoriaMetrics, 2021 [visited on 2021-12-09]. Available from: https://docs.victoriametrics.com/Single-server-VictoriaMetrics.html.

33. SZYPULKA, Rafal. *Monitoring of multiple OpenShift clusters with VictoriaMetrics* [online]. San Francisco: A Medium Corporation, 2021 [visited on 2021-12-09]. Available from: https://medium.com/ibm-garage/monitoring-of-multiple-openshift-clusters-with-victoriametrics-d4f0979e2544.

34. WITZENMAN, Matthew. *Monitoring OpenShift: Three Tools for Simplification* [online]. Raleigh: Red Hat, Inc., 2017 [visited on 2021-12-09]. Available from: https://cloud.redhat.com/blog/monitoring-openshift-three-tools.

35. *Plans to fit your scale* [online]. New York City: Datadog, 2021 [visited on 2021-12-09]. Available from: https://www.datadoghq.com/pricing/?product=infrastructure%5C#infrastructure.

36. *Kubernetes Operational View* [online]. Codeberg, 2020 [visited on 2021-12-09]. Available from: https://codeberg.org/hjacobs/kube-ops-view/src/branch/main/README.md.

37. *Open Source Cloud Computing Infrastructure - OpenStack* [online]. OpenStack, 2021 [visited on 2021-12-09]. Available from: https://www.openstack.org/.

38. *IaaS (Infrastructure-as-a-Service)* [online]. Armonk (New York): IBM Cloud Education, 2019 [visited on 2021-12-09]. Available from: https://www.ibm.com/cloud/learn/iaas.

39. *IaaS vs. PaaS vs. SaaS* [online]. Armonk (New York): IBM Cloud Education, 2021 [visited on 2021-12-09]. Available from: https://www.ibm.com/cloud/learn/iaas-paas-saas.

40. *OpenStack Compute (nova)* [online]. OpenStack, 2021 [visited on 2021-12-09]. Available from: https://docs.openstack.org/nova/latest/.

41. *OpenStack Block Storage (Cinder) documentation* [online]. OpenStack, 2020 [visited on 2021-12-09]. Available from: https://docs.openstack.org/cinder/xena/.

42. *OpenStack Networking* [online]. OpenStack, 2021 [visited on 2021-12-09]. Available from: https://docs.openstack.org/neutron/xena/admin/intro-os-networking.html.

43. *System architecture — Ceilometer 17.0.0.0rc2.dev2 documentation* [online]. OpenStack, 2020 [visited on 2021-12-09]. Available from: https://docs.openstack.org/ceilometer/xena/admin/telemetry-system-architecture.html.

44. *Telemetry* [online]. OpenStack [visited on 2021-12-09]. Available from: https://wiki.openstack.org/wiki/Telemetry.

45. *Alarms — aodh 13.0.0.0rc2.dev2 documentation* [online]. OpenStack, 2020 [visited on 2021-12-09]. Available from: https://docs.openstack.org/aodh/xena/admin/telemetry-alarms.html.

46. CARBONNELLE, Pierre. *PYPL PopularitY of Programming Language* [online]. c2021 [visited on 2021-12-10]. Available from: https://pypl.github.io/.

47. *What is Quarkus?* [Online]. Raleigh: Red Hat, Inc., 2020 [visited on 2021-12-10]. Available from: https://www.redhat.com/en/topics/cloud-native-apps/what-is-quarkus.

48. *Contexts and Dependency Injection* [online]. Raleigh: QuarkusIO, 2021 [visited on 2021-12-11]. Available from: https://quarkus.io/guides/cdi-reference.

49. *GraalVM for Java* [online]. Austin (Texas): Oracle, c2021 [visited on 2021-12-11]. Available from: https://www.graalvm.org/java/.

50. *Kubernetes & OpenShift Java Client* [online]. Fabric8, 2021 [visited on 2021-12-11]. Available from: https://github.com/fabric8io/kubernetes-client/blob/master/README.md.

51. *Apache Camel* [online]. The Apache Software Foundation, c2004-2021 [visited on 2021-12-11]. Available from: https://camel.apache.org/.

52. *QUERYING PROMETHEUS* [online]. San Francisco: Prometheus, 2021 [visited on 2021-12-11]. Available from: https://prometheus.io/docs/prometheus/latest/querying/basics/.

53. *Documentation | Grafana Labs* [online]. New York: Grafana Labs, 2021 [visited on 2021-12-11]. Available from: https://grafana.com/docs/.