Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Open Informatics, Software engineering

Web integrated development environment in private cloud

Bc. Stanislav Ľaš

Supervisor: Ing. Martin Komárek
May 2022

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Ľaš Stanislav**	Personal ID number:	**466294**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master's thesis details

Master's thesis title in English:

Web integrated development environment in private cloud

Master's thesis title in Czech:

Webové vývojové prostředí v privátním cloudu

Guidelines:

Explore web integrated environments, especially platform Gitpod and its possibilities of use in a private cloud to:
- elimination of the time-consuming setup of a local development environment
- improvement of source code storage security

Iteratively design, implement, deploy, and test the solution that enables:
- automatic cloning of the selected GIT repository during workspace initialization
- the ability to commit changes into the GIT repository from the web IDE (automatic user context passing)
- display the relevant services and applications like PostgreDB, Kafka, Redis, etc. for the given workspace
- integration with runtime environments so that the user can consume previously deployed services

Bibliography / sources:

Gitpod-WebIDE [online]. [cit. 2022-02-08]. Dostupné z: https://www.gitpod.io/
Kubernetes - open source system for automating deployment, scaling, and management of containerized applications. [online]. [cit. 2022-02-08]. Dostupné z: https://kubernetes.io/
Helm - The package manager for Kubernetes [online]. [cit. 2022-02-08]. Dostupné z: https://helm.sh/
Concourse [online]. [cit. 2022-02-08]. Dostupné z: https://concourse-ci.org/

Name and workplace of master's thesis supervisor:

Ing. Martin Komárek   Department of Information Security FIT

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **11.02.2022**   Deadline for master's thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Martin Komárek Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

First, I would like to thank Ing. Martin Komarek, the supervisor of this thesis, for his professional support, patience, and willingness to help to overcome the obstacles to writing this thesis. Next, I would like to thank Ondřej Michalčík from the company Stratox for technical guidance and valuable advice. And last but not least, I would like to thank my family and friends for their support during my studies.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

In Prague, May 20th, 2022

Abstract

This thesis deals with web development environments and their use. The work describes the platforms **Github Codespace** and **Gitpod**, which allow the application to be developed directly in the cloud. The thesis, in detail, describes the integration of these platforms with the test application. This document also contains how to deploy the **Gitpod** on the customer‘s infrastructure using various technologies. The tutorial begins by setting up a domain and generating the certificates needed for the platform properly run. Subsequently, it iteratively presents deployment to different clusters. The output of this work is an example that allows the installation of **GitLab** and **Gitpod** on one computer. The platforms are integrated immediately after installation, so you do not need to create a connection. The **Terraform** is used to create the infrastructure and install the platforms, so the deployment is automatic.

Keywords: thesis, virtual workspace, certificates, integration, SSL, Docker, Docker-desktop, Git-crypt, Kubernetes, IDE, Cloud-based Development, Gitpod, Github Codespaces, Kubernetes, k3s, k3d, Terraform, HTTPS, Domain, Git, Github, Gitlab, Helm, DNS, reverse proxy, self-hosted, self-managed, deployment

Supervisor: Ing. Martin Komárek
komarem@fel.cvut.cz

Author: Bc. Stanislav Ľaš
lasstani@fel.cvut.cz
stanislav.las@gmail.com

Abstrakt

Táto diplomová práca sa zaoberá webovými vývojovými prostrediami a ich použitím. V práci sú popísané platformy **Github Codespace** a **Gitpod**, ktoré umožňujú vývoj aplikácie priamo v cloude. Práca podrobne popisuje integráciu týchto platforiem s testovacou aplikáciou. Tento dokument taktiež obsahuje návod ako nasadiť platformu **Gitpod** na vlastnej infraštruktúre pomocou rôznych technológií. Návod začína nastavením domény a vygenerovaním certifikátov potrebných na správny chod platformy. Následne je iteratívnym spôsobom prezentované nasadzovanie do rôznych klastrov. Výstupom tejto práce je príklad, ktorý umožňuje inštaláciu platforiem **GitLab** a **Gitpod** na jednom počítači. Platformy sú po inštalácií ihneď integrované, takže nie je potrebné vytvárať prepojenie. Na vytvorenie infraštruktúry a inštaláciu platforiem je použitý nástroj **Terraform**, ktorý umožňuje, že je celý tento proces automatický.

Klíčová slova: záverečná práca, virtuálne pracovné prostredie, certifikáty, integrácia, nasadenie, SSL, Docker, Docker-desktop, Git-crypt, Kubernetes, IDE, Cloud-based Development, Gitpod, Github Codespaces, Kubernetes, k3s, k3d, Terraform, HTTPS, Domain, Git, Github, Gitlab, Helm, DNS, reverse proxy

Překlad názvu: Webové vývojové prostředí v privátním cloudu

Contents

Figures

Tables

Chapter 1
Introduction

The work of a developer should be creative and original. Therefore, we try to automate the whole development process as much as possible. The evidence is the **CI/CD pipeline**[1], which can automatically run different types of tests, build an application, and even deploy it to the server. However, this process begins after changes are committed to the version control system. Before creative work, there is the process, which is not automated yet. It is a constantly recurring setting up the development environment, in which it is necessary to take into account the operating system of the developer and many other factors. We often hear the phrase, *It works on my machine.* But it does not work on a colleague's machine and not even on the test system. The new style of development called **Cloud-based Development**[**1**] is trying to change that. This style is based on moving the whole development process to the **cloud**, where the developer simply logs in and can start work. The advantage is that all calculations take place on the **cloud**, and access is implemented through a web browser so the developer can work on any device, such as a tablet or mobile phone. Developers working on the project have the same operating system accessible after connecting to the **cloud** system, which simplifies setting up the environment. As long as we provide all developers with the same operating system, why not create a configuration file in which we define the development environment for each application. And this is the biggest advantage of **Cloud-based Development**. The developer creates a configured development environment with one click that can build and run an application. And without further configuration, the developer can begin to develop.

The goal of this thesis is to get familiar with **Cloud-based Development** platforms. And install, configure, and maintain one in a private **cloud**. The two major platforms on the market that currently deal with this style are **Gitpod**[**3**] and **GitHub Codespaces**[**11**]. **Gitpod** has been on the market since 2020, while the beta version of **GitHub Codespaces** got released in August 2021. **Gitpod** is available to public and private repositories for fifty hours a month complimentary. It is an open-source project, and therefore developers involved in the development can request unrestricted access. **Github Codespaces** is currently only available to organizations with

[1]Continuous integration/continuous delivery pipeline

the **Github Team**[2] or **Github Enterprise Cloud**[3] subscription. However, this should change soon because, according to CEO Nat Friedman, access for individuals should be launched by the end of 2021[2]. Nonetheless, there is still no access for individuals yet.

[2]https://docs.github.com/en/get-started/onboarding/getting-started-with-github-team

[3]https://docs.github.com/en/get-started/onboarding/getting-started-with-github-enterprise-cloud

Chapter 2

Gitpod

Gitpod[3] is a platform based on container development, which focuses on developer experience. The platform provides an integrated workspace stored in the cloud, and it allows the user to develop immediately through the web browser on a variety of devices. The workspace configuration is stored in the configuration file so developers can easily create a new workspace for every single task.

Ready to code. A lot of time is lost during the development because of the setting up of the environment on the developer‘s computer. Local development environments are very fragile, and they can be easily corrupted while working on multiple projects at a time. Furthermore, building and downloading dependencies of an application is also a time-consuming task.

In contrast, preparation of an environment with **Gitpod** is elementary. Instead of downloading code developer opens a virtual workspace, where all dependencies are already poised, and the developer can start work creatively. Dependencies are prepared on the bases of configuration, which is stored as the code. Finally, when work is done workspace can be deleted, and the developer does not have to worry about anything else.

Cloud as a local computer. A Virtual workspace contains everything that a developer may need during the development. Workspace is running **lightweight Linux** with editor **VS Code**[1], so the developer has access to the **Unix shell**[2]. Moreover, within the creation of the workspace source codes of the application are downloaded. Workspaces are well customizable, and they can even be preconfigured and optimized on a team, project, or individual level. Additionally, **Gitpod** is compatible with the biggest **git-based** platforms like **Github**[3], **Gitlab**[4], and **Bitbucket**[5].

Velocity. **Gitpod** can start a virtual workspace in terms of seconds. It allows developers to begin work immediately from any **git context**[4], anytime, and

[1]https://code.visualstudio.com/
[2]https://www.tutorialspoint.com/unix/unix-what-is-shell.htm
[3]https://github.com/
[4]https://about.gitlab.com/
[5]https://bitbucket.org/

from any device. The creation of a new workspace is elementary. Moreover, the platform provides the possibility to share the workspace with other team members.

Readiness. **Gitpod** continuously compiles all main branches similar to a **CI server**[6]. When some change is committed into a git repository, it starts building dependencies even before a new virtual workspace is created. The workspace is created almost immediately because dependencies are in place and the application is already to built.

Security. We live in the world of remote connection, where copies of the source code stored on countless insecure devices and networks is not good practice. With **Gitpod**, the code is stored on the **cloud**, and it is accessible only through a secure connection of a web browser. Furthermore, it is possible to access the code from a variety of devices like mobile phones or tablets, and work on all devices is the same as on a classic personal computer. Everything the device needs for the connection to the workspace is a web browser and access to the internet.

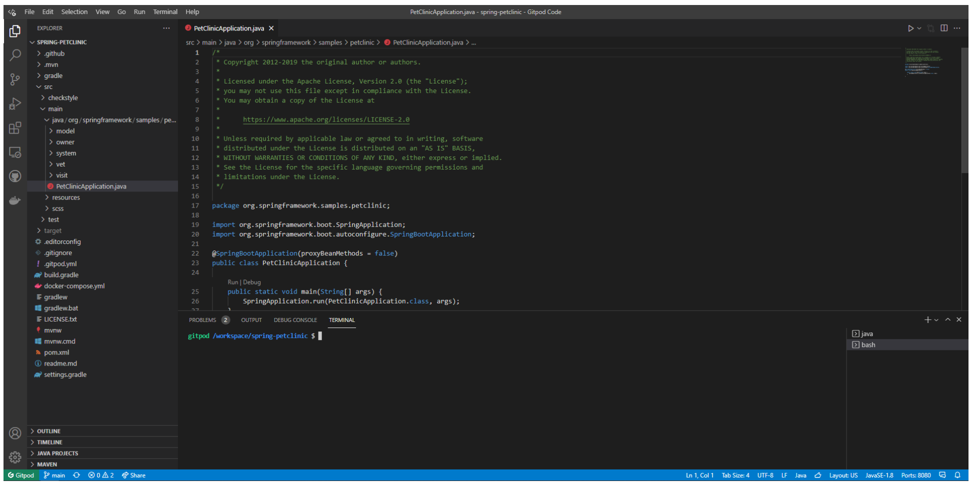
2.1 Integration

To get more familiar with **Gitpod** I decided to integrate with the project **Pet Clinic**[7]. The project is a simple application that demonstrates the **Spring framework**[8]. First, I forked source codes into my personal **Github** (**https://github.com/stanislavlas/spring-petclinic**), and then I followed steps from official **Gitpod** documentation[6].

2.1.1 My first workspace

The best way to configure **Gitpod** is by using **Gitpod**[6]. So I navigated my web browser to the project repository, and in the address bar, I prefixed the entire URL with *gitpod.io/#*. So in my case URL looked like *gitpod.io/#https://github.com/stanislavlas/spring-petclinic*. It redirected me to the **Gitpod** side, where I had to allow a connection between **Github** and **Gitpod** and install the **Gitpod bot** into the repository. After these steps, my first virtual workspace spun up. In the picture[2.1] is what the workspace looks like. It has an integrated editor **VS Code**, and on the left side is a field with standard editors functionality like file explorer and source control manager. On the bottom is opened terminal which provides control of the workspace through **bash**. Moreover, **Gitpod** automatically detected application written in the **Java** and offered me appropriate extensions to the editor. All dependencies were downloaded and when I started the application, workspace proposed to open the application in the new window in the browser.

[6]Continue integration server
[7]https://github.com/spring-projects/spring-petclinic
[8]https://spring.io/



```java
/*
 * Copyright 2012-2019 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.samples.petclinic;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(proxyBeanMethods = false)
public class PetClinicApplication {

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(PetClinicApplication.class, args);
```

```
gitpod /workspace/spring-petclinic $
```

Figure 2.1: Virtual workspace

I could work with an application similar to running it on a local machine in the new window.

2.1.2 Prebuilds

In the previous section, **Gitpod** created a virtual machine and cloned a git repository into it. There is still a need to download dependencies and build the application manually. I created a *.gitpod.yml* configuration file at the project root to get the most out of **Gitpod**. The file provides instructions to **Gitpod** on how to build and prepare the development environment specifically for a project. The config file is created automatically by the command *gp init* at the project root. In the file[2.1.2], there are two sections. One for the definition of which tasks should run and when, and another for the port forwarding definition.

```yaml
# List the start up tasks. Learn more
# https://www.gitpod.io/docs/config-start-tasks/
tasks:
  - init: ./mvnw package
    command: java -jar target/*.jar

# List the ports to expose. Learn more
# https://www.gitpod.io/docs/config-ports/
ports:
  - port: 8080
    onOpen: open-preview
```

Tasks. This part includes a statement, of which jobs will run before, during, or after workspace creation. Jobs that are supposed to be running first when a workspace is created are defined in the section *before*. These tasks are usually for setting up a terminal or installing global dependencies of a project. Section *init* enunciates commands for building an application, downloading dependencies, and time-consuming tasks. It is used for **prebuilds** and allows **Gitpod** to start the workspace in seconds. The section *command* defines what happens after a workspace is created. For example, start the application and open it in the browser. The following figure[2.2] illustrates how these sections are triggered during the start of workspace with and without prebuilds enabled. In the integration, I used *./mvnw package* in the *init* section for

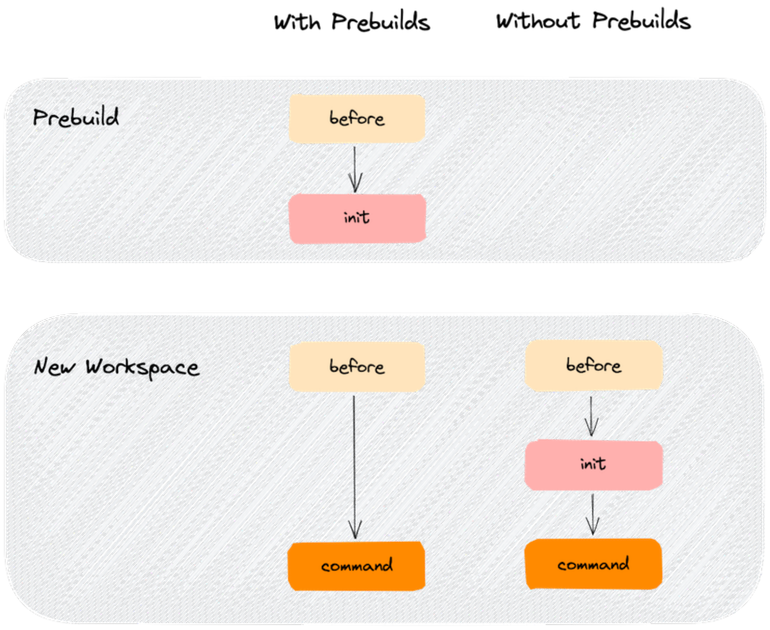
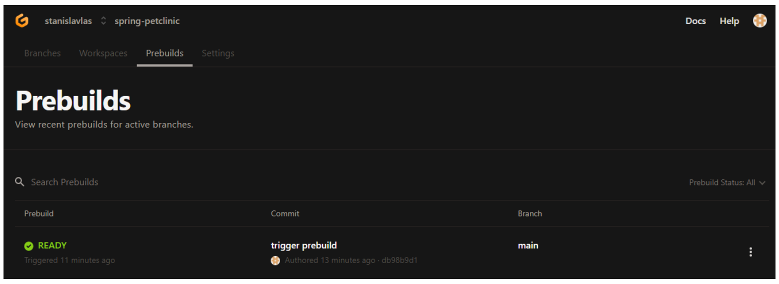


Figure 2.2: Prebuilds[5]

building the application and *java -jar target/*.jar* in the *command* section for starting the application.

Ports. When the project starts a service that listens on a given port, **Gitpod** automatically serves traffic to this port. Additionally, when **Gitpod** detects the available port, it performs the defined task. In our case, **open-preview**, which means it opens the application on a new tab.

At this point, **Gitpod** is configured to be able to perform **prebuild**. **Prebuild** is ordinarily triggered after a change in the repository, by the standard in the main branches. Modification of this manner is possible in the configuration file, but for the integration, I will use default behavior. I created a small change and committed it to the repository, and it triggered **prebuild**, which successfully finished[2.3]. After a successful **prebuild**, I created a new workspace, it started in a few moments, and it opened the running application on a new tab.



Prebuilds

View recent prebuilds for active branches.

Prebuild	Commit	Branch
READY Triggered 11 minutes ago	trigger prebuild Authored 13 minutes ago · db98b9d1	main

Figure 2.3: Successful prebuild

Chapter 3

GitHub Codespaces

Github Codespaces[7] is a virtual development environment stored in the **cloud**. Environment configuration is stored in configuration files in a repository, which creates a repeatable codespace configuration for all users of your project. **Github** provides a variety of virtual machines for running **Codespaces**. Connection to the **Codespace** is provided by a web browser or locally using **VS Code**.

3.1 Codespaces lifecycle

Creation. **Codespace** can be created in several ways:

- implementation of a new feature from the repository
- exploration of work-in-progress from pull request[1]
- investigation of a bug at a specific point at the time from a commit in the repository history
- **Visual Studio Code**

During the creation of a new **codespace** some steps happen in the background before it is available:

1. virtual machine and storage are assigned to the **codespace**
2. the container is created
3. connecting to the **codespace** is established
4. post-creation setup

Codespace as a temporary entity. **Codespace** is considered a temporary entity that will disappear when the work is done. In this case, changes need to be regularly committed to the repository to make sure each new functionality is in the **git**. The maximum number of running **codespaces** is ten. Therefore, if the number is reached, it is necessary to delete one of the old **codespaces** to create a new one.

[1]https://docs.github.com/en/pull-requests

Codespace as a long-running entity. We can understand **codespace** as a long-running entity that we always connect to for every new task. It is necessary to download changes from a standard branch regularly to make sure all new features are available. This workflow is similar to working on a local computer.

Saving changes in a codespace. Through the web, editor **codespace** has enabled auto-save. So changes are automatically saved after a few seconds. The **codespace** is available for thirty minutes from the last activity and subsequently shut down. The activity is considered some change in the editor or output of the terminal. During the long-running tasks, the **codespace** will remain open. Modifications are safely stored in the case of shut down, and at the next start of the **codespace**, the state is restored. Changes are not saved automatically during work on a local computer in **VS Code**. It requires some configuration of **VS Code** to enable auto-save of changes.

Stopping and closing a codespace. The way how to shut down a **codespace** is using the command palette in the **VS Code** (*Shift + Command + P* (Mac) or *Ctrl + Shift + P* (Windows)) by entering the command *Codespaces: Stop Current Codespace*. **Codespace** is not shut down when the tab is closed. However, It is automatically shut down after a predefined time (default thirty minutes) When deleting the **codespace**, it detects all uncommitted changes, and the editor prompts the developer to commit or discard the changes.

Access to application in a codespace. Application in a **codespace** that is accessible allows port forwarding, which defines ports accessible outside of **codespace**. Processes running inside a **codespace** can access the application, even without port forwarding. Forwarded ports are not accessible from the internet by default. However, it is possible to configure it and expose it to the organization‘s network or the internet. **Codespace** is stored on the cloud, and it requires an internet connection to access it. It is not possible to work in **codespace** when the connection is lost. On the other hand, all uncommitted changes are stored and in the incoming connection to **codespace**, the changes are restored.

Committing changes. Every **codespace** contains the **git**, which allows working in a **codespace** according to standard **git** workflows. Working with **git** is possible by using a terminal or the interface for resource management integrated into the **VS Code**.

Personalization. The editor allows the installation of various extensions from **Visual Studio Code Marketplace**[2]. Moreover, possibility to synchronize all extensions, settings, themes, and keyboard shortcuts from the local **VS Code**.

[2]https://marketplace.visualstudio.com/vscode

3.2 Dev container

Dev container[8] is an environment, which provides tools necessary for software development. There is a possibility to define own container. However, **codespace** will use the default one with regular tools for development when the container is not defined. The configuration of a container must be stored in a folder *.devcontainer*, which consists of files *devcontainer.json* and *Dockerfile*. The main configuration is stored in the file *devcontainer.json*, which is mandatory. *Dockerfile* is optional, but it defines an image used for the container, and in some scenarios, it might be handy. A reference needs to be added to the *devcontainer.json*. **Codespace** offers several predefined configurations for a specific project type. These configurations are accessible via the pallet of commands in **VS Code** (*Shift + Command + P* (Mac) or *Ctrl + Shift + P* (Windows)) after selecting *Codespaces: Add Development Container Configuration Files <...>*.

3.3 Integration

Codespaces are currently only available for organizations that use **GitHub Team**[?] or **GitHub Enterprise Cloud**[10]. Unfortunately, it is not available for individuals, so I could not try integration on my project. According to the official guide[11], it should be simple. After clicking on the *Code* button in a repository, the **Codespace** tab will appear. This tab contains all created **codespaces** and a button for creating a new one. **VS Code** editor with terminal will appear after the creation of **codespace**. An application can be compiled and run using a terminal or the editor. When **codespace** detects the port on which the application is running, it offers port forwarding and opens the application in a new browser tab.

3.3.1 Dev container configuration

To define a container, we can choose one of the predefined configurations or create a file *.devcontainer.json*. In the example[3.3.1], we can see the *.devcontainer.json* file generated after using a predefined **Java** configuration. At the beginning of the file is the defined name of the container followed by the definition of an application compilation, in our case **Dockerfile** with appropriate arguments. Next, we can see the editor settings such as the type of terminal, home directory for **Java**, and the path to the package manager **Maven**. In other options, we can see the settings of extensions in **VS Code**, port forwarding, and commands to run after **codespace** creation. Finally, the user, which the developer logs into the **codespace**, is defined. By default the user **vscode** is used, but the alternative is to use the **root** user, for example.

```
{
  "name": "Node.js",
  "build": {
    "dockerfile": "Dockerfile",
    "args": { "VARIANT": "14" }
  },

  "settings": {
    "terminal.integrated.shell.linux": "/bin/bash"
  },

  "extensions": [
    "dbaeumer.vscode-eslint"
  ],

  // "forwardPorts": [],

  // "postCreateCommand": "yarn install",

  "remoteUser": "node"
}
```

The **Dockerfile**[**3.3.1**] is also generated from a predefined configuration for the **Java**. As we can see at the beginning, the image on which the container will run is defined. Subsequently, the arguments and the installation of the package manager are defined, followed by the installation of the **Node.js** application.

```
# [Choice] Node.js version: 14, 12, 10
ARG VARIANT="14-buster"
ARG URL="mcr.microsoft.com"
ARG PATH="/vscode/devcontainers/javascript-node:0"
FROM ${URL}${PATH}-${VARIANT}

# [Optional] Uncomment this section to
# install additional OS packages.
# RUN apt-get update && \
#    export DEBIAN_FRONTEND=noninteractive && \
#    apt-get -y install --no-install-recommends \
#    <your-package-list-here>

# [Optional] Uncomment if you want to
# install an additional version of node using nvm
# ARG EXTRA_NODE_VERSION=10
# RUN su node -c "source /usr/local/share/nvm/nvm.sh && \
#    nvm install \
#    ${EXTRA_NODE_VERSION}"

# [Optional] Uncomment if you want
# to install more global node modules
RUN su node -c "npm install -g <your-package-list-here>"
```

Chapter 4
Gitpod Self-hosted

Gitpod supports an installation on the customer‘s infrastructure[12], which must include the distribution of **Kubernetes** platform. Supported distributions are **Amazon Elastic Kubernetes Service**, **Google Kubernetes Engine**, **k3s**, and **Microsoft Azure Kubernetes Service**. **Gitpod** also requires a domain name resolvable by a **DNS server** in infrastructure and trusted **HTTPS certificates** for **SSL communication**.

4.1 Domain name

Since **Gitpod** requires a domain resolvable by the DNS, I bought the domain *test-gitpod.com* on side https://www.godaddy.com/. I associated it with the public IP address of my local network by adding **A** record into DNS records in DNS management of the domain. As a quick test of the connection with my network, I ran a simple server on my computer with *python3 -m http.server 8080*, which started service on *localhost:8080.* When I tried to hit *test-gitpod.com:8080* request failed because *test-gitpod.com* refused to connect. The reason for failure was that I did not allow connection on my router which is possible by **port forwarding**[1]. My TP-Link router supports **port forwarding** by the feature called **"Virtual servers"**. The feature requires an IP address of the computer in the local network, external and internal port of the service, service type and, the protocol. After I set up **port forwarding** on port **8080** I could reach the server from outside of my home network by using the domain *test-gitpod.com.* **Gitpod** launches services and workspaces on additional subdomains, it also needs two wildcard domains, so I created all DNS records 4.1 in DNS management of the domain.

```
your-domain.com
*.your-domain.com
*.ws.your-domain.com
```

[1]https://learn.g2.com/port-forwarding

4.2 Certificates

Gitpod requires HTTPS certificates. There is no hard requirement on any certificate authority, but the recommendation is to use the **ACME certificate issuer**[2] to renew and install certificates automatically[25]. I used **certbot**[3] to get certificates from **Let‘s Encrypt**[4] by command on 4.2. Command runs the **certbot** image in docker container with attached volumes where certificates will be saved followed by my email. I used the **DNS challenge**[5] with **manual** domain owner confirmation. It means the certbot asks to deploy randomly generated **TXT** records into DNS management of the domain and then provides certificates. The last three arguments are domains, which must be included in certificates.

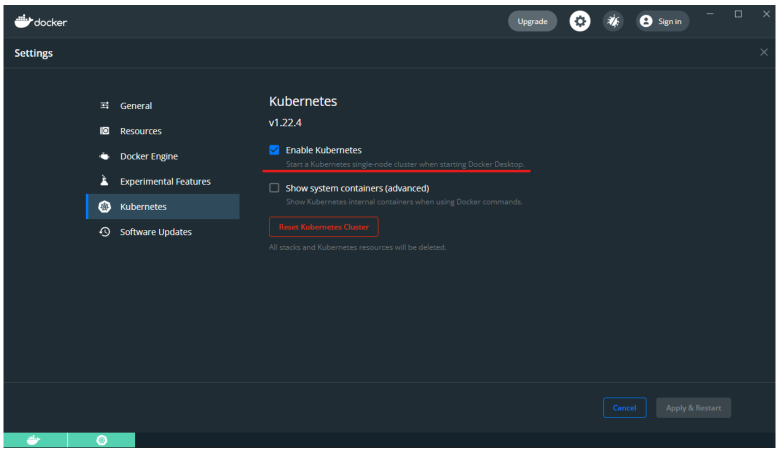
```
sudo docker run -it --rm --name certbot \
    -v $WORKDIR/etc:/etc/letsencrypt \
    -v $WORKDIR/var:/var/lib/letsencrypt \
        certbot/certbot certonly \
            -v \
            --email stanislav.las@gmail.com \
            --manual \
            --preferred-challenges=dns \
            --agree-tos \
            -d test-gitpod.com \
            -d *.test-gitpod.com \
            -d *.ws.test-gitpod.com
```

4.3 Kubernetes

4.3.1 Kubernetes in Docker-desktop

First, I tried to make **Gitpod** work on my computer in the **Docker[13]**. I used **Docker-desktop[14]** on the operation system **Windows 10** and **WSL[16]**. **Docker-desktop** contains a standalone **Kubernetes** server[15] on a local computer. The server is not configurable, contains one node, and is intended for testing purposes. Enabling of **Kubernetes** support is placed in the settings of **Docker-desktop** in *Preferences > Kubernetes*[4.1]. Installation can be validated by the tool *kubectl get nodes*[4.2], which returns all nodes in the cluster. The **Kubernetes** command-line tool, **kubectl[17]**, allows users to run commands against **Kubernetes** clusters. It needs to have a properly configured context pointing into the correct **Kubernetes** cluster. Installation of **Gitpod** requires a configuration file, so I created

[2]https://caddyserver.com/docs/automatic-https
[3]https://eff-certbot.readthedocs.io
[4]https://letsencrypt.org/
[5]https://letsencrypt.org/docs/challenge-types/



Kubernetes

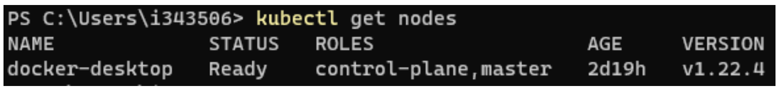
v1.22.4

[✓] Enable Kubernetes
Start a Kubernetes single-node cluster when starting Docker Desktop.

[ ] Show system containers (advanced)
Show Kubernetes internal containers when using Docker commands.

Reset Kubernetes Cluster
All stacks and Kubernetes resources will be deleted.

Figure 4.1: Enabling **Kubernetes** support



```
PS C:\Users\i343506> kubectl get nodes
NAME             STATUS   ROLES                  AGE     VERSION
docker-desktop   Ready    control-plane,master   2d19h   v1.22.4
```

Figure 4.2: kubectl output

*values.custom.yaml*4.3.1. These values are used for configuration tools, such as **RabbitMQ**[6] and **Minio**[7], which are part of **Gitpod** installation. In my case, the only condition was that keys for **Minio** must have the correct format. Therefore, I generated it by the command *openssl rand -hex 20*[8].

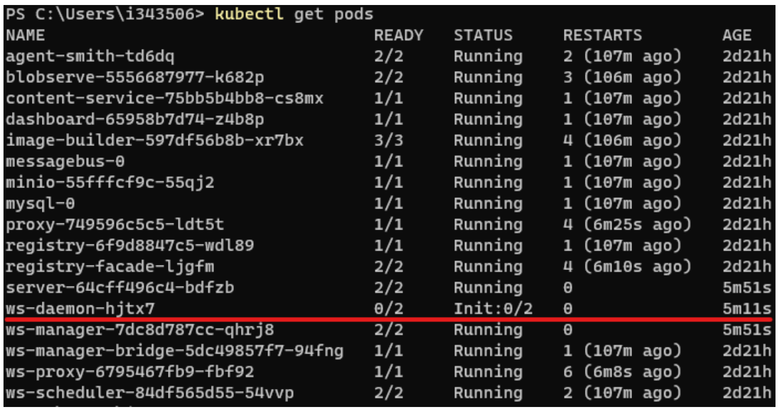
```
docker-registry:
  authentication:
    username: gitpod
    password: your-registry-password
rabbitmq:
  auth:
    username: your-rabbitmq-user
    password: your-secret-rabbitmq-password
minio:
  accessKey: your-random-access-key
  secretKey: your-random-secret-key
```

Next, I installed **Gitpod** by **Kubernetes** deployment manager **helm**[18] with these commands[4.3.1].

```
helm repo add gitpod.io https://charts.gitpod.io
helm repo update
helm install -f values.custom.yaml gitpod \
                gitpod.io/gitpod --version=0.10.0
```

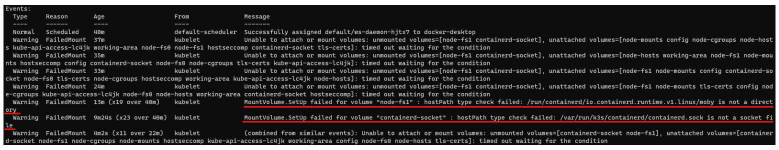
[6]https://www.rabbitmq.com/
[7]https://min.io/
[8]https://www.wolfssl.com/docs/

It installed **Gitpod** on the local **Kubernetes cluster**, but some pods were not in the **RUNNING** state. The reason was that I did not provide SSL certificates, so I added them into the cluster as a secret. At this point, almost all pods were running correctly[4.3], except *ws-daemon*, which was failing on the error[4.4]. The description of the error is on the official



```
PS C:\Users\i343506> kubectl get pods
NAME                                 READY   STATUS      RESTARTS       AGE
agent-smith-td6dq                    2/2     Running     2 (107m ago)   2d21h
blobserve-5556687977-k682p           2/2     Running     3 (106m ago)   2d21h
content-service-75bb5b4bb8-cs8mx     1/1     Running     1 (107m ago)   2d21h
dashboard-65958b7d74-z4b8p           1/1     Running     1 (107m ago)   2d21h
image-builder-597df56b8b-xr7bx       3/3     Running     4 (106m ago)   2d21h
messagebus-0                         1/1     Running     1 (107m ago)   2d21h
minio-55fffcf9c-55qj2                1/1     Running     1 (107m ago)   2d21h
mysql-0                              1/1     Running     1 (107m ago)   2d21h
proxy-749596c5c5-ldt5t               1/1     Running     4 (6m25s ago)  2d21h
registry-6f9d8847c5-wdl89            1/1     Running     1 (107m ago)   2d21h
registry-facade-ljgfm                2/2     Running     4 (6m10s ago)  2d21h
server-64cff496c4-bdfzb              2/2     Running     0              5m51s
ws-daemon-hjtx7                      0/2     Init:0/2    0              5m11s
ws-manager-7dc8d787cc-qhrj8          2/2     Running     0              5m51s
ws-manager-bridge-5dc49857f7-94fng   1/1     Running     1 (107m ago)   2d21h
ws-proxy-6795467fb9-fbf92            1/1     Running     6 (6m8s ago)   2d21h
ws-scheduler-84df565d55-54vvp        2/2     Running     2 (107m ago)   2d21h
```

Figure 4.3: **Gitpod pods**



```
Events:
  Type     Reason       Age                  From               Message
  ----     ------       ----                 ----               -------
  Normal   Scheduled    40m                  default-scheduler  Successfully assigned default/ws-daemon-hjtx7 to docker-desktop
  Warning  FailedMount  37m                  kubelet            Unable to attach or mount volumes: unmounted volumes=[node-fs1 containerd-socket], unattached volumes=[node-mounts config node-cgroups node-host
s kube-api-access-lc4jk working-area node-fs0 node-fs1 hostseccomp containerd-socket tls-certs]: timed out waiting for the condition
  Warning  FailedMount  35m                  kubelet            Unable to attach or mount volumes: unmounted volumes=[node-fs1 containerd-socket], unattached volumes=[node-hosts working-area node-fs1 node-mou
nts hostseccomp config containerd-socket node-fs0 node-cgroups tls-certs kube-api-access-lc4jk]: timed out waiting for the condition
  Warning  FailedMount  33m                  kubelet            Unable to attach or mount volumes: unmounted volumes=[node-fs1 containerd-socket], unattached volumes=[node-fs1 node-mounts config containerd-so
cket node-fs0 tls-certs node-cgroups hostseccomp working-area kube-api-access-lc4jk node-hosts]: timed out waiting for the condition
  Warning  FailedMount  24m                  kubelet            Unable to attach or mount volumes: unmounted volumes=[node-fs1 containerd-socket], unattached volumes=[node-fs1 node-mounts tls-certs config nod
e-cgroups kube-api-access-lc4jk node-fs0 node-hosts working-area containerd-socket hostseccomp]: timed out waiting for the condition
  Warning  FailedMount  13m (x19 over 40m)   kubelet            MountVolume.SetUp failed for volume "node-fs1" : hostPath type check failed: /run/containerd/io.containerd.runtime.v1.linux/moby is not a direct
ory
  Warning  FailedMount  9m24s (x23 over 40m)  kubelet           MountVolume.SetUp failed for volume "containerd-socket" : hostPath type check failed: /var/run/k3s/containerd/containerd.sock is not a socket fi
le
  Warning  FailedMount  4m2s (x11 over 22m)  kubelet            (combined from similar events): Unable to attach or mount volumes: unmounted volumes=[containerd-socket node-fs1], unattached volumes=[container
d-socket node-fs1 node-cgroups node-mounts hostseccomp kube-api-access-lc4jk working-area config node-fs0 node-hosts tls-certs]: timed out waiting for the condition
```

Figure 4.4: Error in *ws-daemon*

web of the **Gitpod**[19]. The suggested solution is to configure paths to the **containerd[22]** and upgrade the installation. At this point, I ran into the problem because the cluster runs on **Docker-desktop**, and it does not allow trivial configuration. I could not change the paths, so I decided to switch to the **Linux** to prevent similar issues.

4.3.2 Microk8s

As mentioned in the documentation, **Gitpod** requires **Kubernetes** installed on the machine, so I decided to use **microk8s[20]**. Its installation is elementary and quick. I followed the installation guide, which uses installation manager **snap**. A snap is a bundle of applications and their dependencies that works without modification across many different Linux distributions[24]. Installation of **Microk8s** is by command *sudo snap install microk8s –classic*, and it is possible to check if the cluster is ready by *microk8s status –wait-ready*. When the cluster is ready, it needs to enable some services with *microk8s enable dashboard dns registry istio*. After the proper installation of the cluster, the command *microk8s kubectl get all –all-namespaces* should return all resources running on the cluster. Also, there is a way to enable the dashboard with *microk8s dashboard-proxy*, but I think it is better to use the command line for configuration. **Microk8s** provides a simple way to pause and start the cluster by *microk8s start* and *microk8s stop*, respectively. Unfortunately, I was not able to make **Gitpod** work on the **microk8s**. All pods went into the **RUNNING** state, but I could not access service on the *test-gitpod.com*. I am not sure what the problem was, but I decided not to continue with the **microk8s** and use **k3s** instead, which is listed as a supported by **Gitpod**.

4.3.3 K3s

K3s[29] is lightweight **Kubernetes** that is production-ready, easy to install, half the memory, all in a binary less than 100MB[29] in size. **K3s** installation is by the command *curl -sfL https://get.k3s.io | sh -*, and to verify the cluster works, I ran *k3s kubectl get node*. Running *k3s kubectl...* every time is not much pleasure, so I copied k3s kubectl config into local kube configuration by 4.3.3 so I can run *kubectl get nodes* now.

```
sudo k3d kubeconfig get gitpod > $HOME/.kube/config
```

When I installed **Gitpod** on k3s proxy pod could not start because of an error: 0/1 nodes are available: 1 node(s) didn't have free ports for the requested pod ports. This error is caused by **traefik proxy**[9] since **Gitpod** has its proxy. Therefore I had to uninstall it, which consists of a few steps[26]:

1. Remove traefik helm chart resource: *kubectl -n kube-system delete helmcharts.helm.cattle.io traefik*
2. Stop the k3s service: *sudo service k3s stop*
3. Edit service file: *sudo nano /etc/systemd/system/k3s.service* and add this line to **ExecStart**: *–no-deploy traefik*
4. Reload the service file: *sudo systemctl daemon-reload*
5. Remove the manifest file from auto-deploy folder: *sudo rm /var/lib/rancher/k3s/server/manifests/traefi*
6. Start the k3s service: *sudo service k3s start*

It is possible to disable the deployment of **Traefik** in the installation by adding the k3s flag into the installation command: *curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="–disable=traefik" sh*

4.4 Gitpod installation

Helm. This section describes how to install **Gitpod** on any **Kubernetes** cluster using **Helm**. The chart for releases resides in the **Helm** repository[charts.gitpod.io], and the source of the charts is in the git repository[https://github.com/gitpod-io/gitpod/blob/main/chart/]. I created configuration file *values.custom.yaml[4.4]* and replaced the key/secrets with

[9]https://traefik.io/

random values. The values are used for internal communication inside the application.

```
docker-registry:
  authentication:
    username: gitpod
    password: your-registry-password
rabbitmq:
  auth:
    username: your-rabbitmq-user
    password: your-secret-rabbitmq-password
minio:
  accessKey: your-random-access-key
  secretKey: your-random-secret-key
```

The next step was to add charts repo into the **Helm**, update **Helm** repositories, and install **Gitpod** using these commands 4.4.

```
helm repo add gitpod.io https://charts.gitpod.io
helm repo update
helm install -f values.custom.yaml gitpod \
                gitpod.io/gitpod --version=0.10.0
```

At this point, **Gitpod** was deployed, but some pods were not starting correctly. The reason is that I did not set up the domain and certificates for installation. Configuration of the domain[4.1] is in *values.custom.yaml[4.4]* by adding the *test-gitpod.com* in *hostname* and my public IP address in *loadBalancerIP*.

```
hostname: <your-domain.com>
components:
  proxy:
    loadBalancerIP: <your-IP>
```

Generation of the certificates was in 4.2. I moved them into *secrets/https-certificates* and renamed them to *tls.crt* and *tls.key*. I also generated *dh-params.pem* and ran the **kubectl** command to create Kubernetes secret **https-certificates** from folder *secrets/https-certificates*. When I set up the domain and certificates I had to upgrade the **Gitpod** installation. It was done by following commands 4.4.

```
# Generate the dhparams.pem
openssl dhparam -out \
  secrets/https-certificates/dhparams.pem 2048

# Create new secret with name httpsCertificates
kubectl create secret generic https-certificates \
  --from-file=secrets/https-certificates

# Upgrade installation of gitpod
helm upgrade --install -f values.custom.yaml \
  gitpod gitpod.io/gitpod --version=0.10.0
```

After this setup, all pods were in a **RUNNING** state, but I could not access the **Gitpod** dashboard. The problem was that **Gitpod** runs on port **443**, but my router did not forward the port. I added port forwarding for port **443**, and at this point, I could open the dashboard and run workspaces on my **Self Hosted Gitpod**. Unfortunately, this installation is currently deprecated, so I had to use a different approach which is using an installer.

Installer. The current way to install **Gitpod** on your **Kubernetes** cluster is using the **Gitpod installer**. The installer is currently available only for **Linux**. I used **k3s** as a **Kubernetes** cluster, which I installed in 4.3.3. The first step was to download the latest version of the installer from *https://github.com/gitpod-io/gitpod/releases/download/2022.01/gitpod-installer-linux-amd64*. Next, I installed the binary and tested it to ensure the version I installed was up-to-date. It is possible to generate the base config file[4.4] used in the installation with the installer. For all these steps, I used the following commands[4.4].

```
# Download the latest release with the command
curl -fsSLO https://github.com/gitpod-io/gitpod/ \
    releases/download/2022.01/gitpod-installer-linux-amd64
# Download the checksum file
curl -fsSLO https://github.com/gitpod-io/gitpod/releases \
    /download/2022.01/gitpod-installer-linux-amd64.sha256
# Validate the binary against the checksum file
echo "$(<gitpod-installer-linux-amd64.sha256)" \
    | sha256sum --check
# Install the binary
sudo install -o root -g root gitpod-installer-linux-amd64 \
    /usr/local/bin/gitpod-installer
# Test to ensure the version you installed it up-to-date
gitpod-installer version
# Generate the base config
gitpod-installer init > gitpod.config.yaml
```

I added the domain *test-gitpod.com* and the path where **Gitpod** containers are stored in this file[4.4] because the path is different in **k3s**. The path is found by running *mount | grep rootfs* on the node.

```
apiVersion: v1
authProviders: []
blockNewUsers:
  enabled: false
  passlist: []
certificate:
  kind: secret
  name: httpsCertificates
containerRegistry:
  inCluster: true
  s3storage: null
database:
  inCluster: true
domain: test-gitpod.com
imagePullSecrets: null
jaegerOperator:
  inCluster: true
kind: Full
metadata:
  region: local
objectStorage:
  inCluster: true
observability:
  logLevel: info
repository: eu.gcr.io/gitpod-core-dev/build
workspace:
  resources:
    requests:
      cpu: "1"
      memory: 2Gi
  runtime:
    containerdRuntimeDir: \
    /run/k3s/containerd/io.containerd.runtime.v2.task/k8s.io
    containerdSocket: /run/k3s/containerd/containerd.sock
    fsShiftMethod: fuse
```

Before installation, I ran scripts for validation of the config file[4.4].

```
# Checks the validity of the configuration YAML
gitpod-installer validate config \
    --config gitpod.config.yaml

# Checks that your cluster is ready to install Gitpod
gitpod-installer validate cluster --kubeconfig \
    ~/.kube/config --config gitpod.config.yaml
```

The second check failed because I did not have some required dependencies set up correctly, which can be seen in the output of validation[4.4].

```
{
  "status": "ERROR",
  "items": [
    {
      "name": "Linux kernel version",
      "description": "all cluster nodes run Linux 5.4.0-0",
      "status": "OK"
    },
    {
      "name": "containerd enabled",
      "description": "all cluster nodes run containerd",
      "status": "OK"
    },
    {
      "name": "Kubernetes version",
      "description": "all cluster nodes run k8s 1.21.0-0",
      "status": "OK"
    },
    {
      "name": "affinity labels",
      "description": "Affinity labels not present in cluster",
      "status": "ERROR",
      "errors": [
        {
          "message": "gitpod.io/workload_ide",
          "type": "ERROR"
        },
        {
          "message": "gitpod.io/workload_workspace_services",
          "type": "ERROR"
        },
        {
          "message": "gitpod.io/workload_workspace_regular",
          "type": "ERROR"
        },
        {
          "message": "gitpod.io/workload_workspace_headless",
          "type": "ERROR"
        },
        {
          "message": "gitpod.io/workload_meta",
          "type": "ERROR"
        }
      ]
    },
    {
      "name": "cert-manager installed",
      "description": "cert-manager is installed",
      "status": "ERROR",
      "errors": [
        {
          "message": "cannot find the cert-manager",
          "type": "ERROR"
        }
      ]
    },
    {
      "name": "Namespace exists",
      "description": "ensure that the namespace exists",
      "status": "OK"
    },
    {
      "name": "https-certificates is present and valid",
      "description": "cannot find the https-certificates",
      "status": "ERROR",
      "errors": [
        {
          "message": "secret https-certificates not found",
          "type": "ERROR"
        }
      ]
    }
  ]
}
```

The installer required **affinity labels** and installation of a **cert-manager**[10], which I installed by command *kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.7.1/cert-manager.yaml*. **Cert-manager** should generate the certificates, but I use **ClouDNS**[11] as DNS manager, which **cert-manager** does not support. Therefore I had to create the **HTTPS certificates** without cert-manager, add them to the file[4.4], and run *kubectl apply -f https-certificates*. The certificate and the key in the file must be encrypted by **Base64**[12] encryption.

[10]https://cert-manager.io/docs/
[11]https://www.cloudns.net/
[12]https://www.base64decode.org/

```
apiVersion: v1
kind: Secret
metadata:
  name: https-certificates
data:
  tls.crt: <cert>
  tls.key: <key>
```

After these steps everything was set up so I rendered *gitpod.yaml* and deploy **Gitpod** by 4.4.

```
# Render the YAML
gitpod-installer render --config \
        gitpod.config.yaml > gitpod.yaml

# Deploy
kubectl apply -f gitpod.yaml
```

In a few minutes, I could access the **Gitpod** dashboard on *test-gitpod.com*. First, I set up integration with **Github** by creating **OAuth App**[44] in **Github** and providing the *client ID* and *client secret* to the **Gitpod**. I also tried to create a new workspace from **Pet Clinic**, which got successfully created. After the build, application started in a new window as I set up in the chapter[2.1.2].

Chapter 5
Gitpod and Gitlab Self-hosted

Requirements for **GitLab** deployed on **Kubernetes** are *kubectl 1.16 or higher* and *helm v3 or higher* installed on the machine. Default Helm chart configuration creates an implementation where all **GitLab** services placing into a cluster similar to **Gitpod** installation. However, it can be configured to point to external stateful storage such as PostgreSQL, Redis, all Non-Git repository storage, or Git repository storage[27]. The **Gitpod** was already running in the **k3s** cluster. So decided to install **GitLab** on the same cluster to another namespace.

5.1 Domain and Certificates

By default, the chart relies on Kubernetes **Service** objects of type **LoadBalancer** to expose **GitLab** services using name-based virtual servers configured with **Ingress** objects. Therefore it needs a domain that will contain records to resolve *GitLab*, *registry* and *minio* to the appropriate IP address[28]. I decided I use ***gitpod****.test-gitpod.com* for **Gitpod** and ***gitlab****.test-gitlab.com* for **GitLab**. So I changed records in the DNS manager for the domain *test-gitpod.com* to contain the following **A** records[5.1].

```
test-gitpod.com
*.test-gitpod.com
*.gitlab.test-gitpod.com
*.gitpod.test-gitpod.com
*.ws.gitpod.test-gitpod.com
```

GitLab should be running with **HTTPS**, which requires TLS certificates. There are few options for obtaining free certificates, but I have already done it using **certbot** for **Gitpod**, so I chose the same approach[4.2] for **GitLab**. I used similar command to the one I already used in 4.2. However in this case, I also included wildcard DNS records for **GitLab**[5.1]. Next, I updated the certificate secret with a new certificate and key, which contains records for **GitLab**.

```
sudo docker run -it --rm --name certbot \
    -v $WORKDIR/etc:/etc/letsencrypt \
    -v $WORKDIR/var:/var/lib/letsencrypt \
        certbot/certbot certonly \
            -v \
            --email stanislav.las@gmail.com \
            --manual \
            --preferred-challenges=dns \
            --agree-tos \
            -d test-gitpod.com \
            -d *.test-gitpod.com \
            -d *.gitlab.test-gitpod.com \
            -d *.gitpod.test-gitpod.com \
            -d *.ws.gitpod.test-gitpod.com

kubectl apply -f https-certificates
```

5.2 Installation on k3s

Once I had all of my configuration options collected, I could get all dependencies and ran the **helm** command for deployment[5.2]. First, I added the **GitLab** repository to the **helm** and updated it. I called this deployment **gitlab**, and I specified the domain, the file name where certificates were stored, and that I do not want to use **cert-manager**[1].

```
helm repo add gitlab https://charts.gitlab.io/
helm repo update
helm install gitlab gitlab/gitlab \
  --set global.hosts.domain=test-gitpod.com \
  --set certmanager.install=false \
  --set global.ingress.configureCertmanager=false \
  --set global.ingress.tls.secretName=https-certificates
```

After a few minutes, some pods were still not in the **RUNNING** state, so I investigated why. My findings were that **GitLab** uses the proxy running on the same port as the proxy in **Gitpod**. Therefore **GitLab** proxy could not start, so I had to find how to run both proxies on the server.

5.3 K3d

K3d[30] is a lightweight wrapper to run **K3s** in the docker container. **K3d** uses a Docker image built from the K3s repository to spin up multiple **k3s** nodes in Docker containers on any machine with Docker installed. Because

[1]https://cert-manager.io/docs/

of that, a single computer can run a various number of **k3s** clusters, with multiple server and agent nodes each, simultaneously[30]. I found the example of how to install both **GitLab** and **Gitpod** on **k3d**[**21**]. The idea is to run two **k3s** clusters, one for **Gitpod**, and another one for **GitLab**, and on top of that, run **Nginx**[2] reverse proxy server. A reverse proxy server is a type of proxy server that typically sits behind the firewall in a private network and directs client requests to the appropriate backend server[31]. The first step was to install **k3d** and create clusters for **Gitpod** and **GitLab**[5.3]. I used different ports for the clusters, so they have not interfered anymore. Additionally, I created the reverse proxy server on port **443** which will be forwarding the communication to the appropriate cluster. I disabled the **Traefik proxy** for both clusters to prevent errors, which I mentioned in the previous chapter[4.3.3]. For the **Gitpod** cluster, I also attached the volume for workspaces. Next, I created a secret for certificates in both clusters, and I used the last command to be able to swap between clusters.

```
# Install K3d
wget -q -O - https://raw.githubusercontent.com/\
    k3d-io/k3d/main/install.sh | bash

# Create Gitlab cluster
sudo k3d cluster create \
    -p 1443:443@loadbalancer \
    --k3s-arg "--disable=traefik@server:0" \
    gitlab

# Create Gitpod cluster
mkdir -p /tmp/workspaces
sudo k3d cluster create \
    -p 2443:443@loadbalancer \
    -v /tmp/workspaces:/var/gitpod/workspaces:shared \
    --k3s-arg "--disable=traefik@server:0" \
    gitpod

# Create secrets for certificates
k3d get kubeconfig gitlab --switch
kubectl apply -f https-certificates
k3d get kubeconfig gitpod --switch
kubectl apply -f https-certificates

# Command to setup kubectl
sudo k3d kubeconfig get <gitpod/gitlab> > $HOME/.kube/config
```

[2]https://www.nginx.com/

5.3.1 Reverse proxy server

The reverse proxy server configuration is in file *default.conf*[5.3.1]. In the configuration file, I created two upstream services for **GitLab** and **Gitpod** where I defined URLs and ports pointing to clusters. I also built up two servers that formulate how forwarding works with certificates and proxy headers. There is a parameter *client_max_body_size* which I had to increase to *100m* for **GitLab** because it requires a bigger body size.

```
map $http_upgrade $connection_upgrade {
    default upgrade;
    ''      close;
}
map $http_upgrade $vs_connection_header {
    default upgrade;
    ''      $default_connection_header;
}


upstream gitlab {
    server gitlab.test-gitpod.com:1443;
}

server {
    listen 443 ssl;
    server_name gitlab.test-gitpod.com
                registry.test-gitpod.com
                minio.test-gitpod.com;

    ssl_certificate /etc/nginx/certs/fullchain.pem;
    ssl_certificate_key /etc/nginx/certs/privkey.pem;

    location / {
        client_max_body_size 100m;

        set $default_connection_header close;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $vs_connection_header;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For \
                            $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Host $host;
        proxy_set_header X-Forwarded-Port $server_port;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_pass https://gitlab;
    }
}


upstream gitpod {
    server gitpod.test-gitpod.com:2443;
}

server {
    listen 443 ssl default_server;
    server_name _;

    ssl_certificate /etc/nginx/certs/fullchain.pem;
    ssl_certificate_key /etc/nginx/certs/privkey.pem;

    location / {
        client_max_body_size 10g;

        set $default_connection_header close;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $vs_connection_header;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For \
                            $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Host $host;
        proxy_set_header X-Forwarded-Port $server_port;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_pass https://gitpod;
    }
}
```

I attached the config file as a volume into the docker container alongside volumes for certificate and key, which I obtained in 5.1, and create a reverse proxy server by 5.3.1.

```
sudo docker run --rm --name nginx-proxy \
-v "$ROOT_DIR/default.conf:/etc/nginx/conf.d/default.conf" \
-v "$ROOT_DIR/fullchain.pem:/etc/nginx/certs/fullchain.pem" \
-v "$ROOT_DIR/privkey.pem:/etc/nginx/certs/privkey.pem" \
-p 0.0.0.0:443:443 -d nginx
```

5.3.2 GitLab

For **GitLab** installation I used the same approach as in 5.2. First, I had to set up *kubectl* to point into the **GitLab** cluster. The installation went relatively smoothly, but I ran into two problems. The first, when I tried to hit *gitlab.test-gitpod.com* it returned **502 - Bad Gateway** status code. I found out the *gitlab.test-gitpod.com:1443* is refusing the connection when I take a look into reverse proxy logs. The reason was that router was not port forwarding **1443**, so I allowed it similar to 4.1. Another problem was the low value of maximum body size, which I solved by increasing the value in the reverse proxy[5.3.1] to *100m*. At this point, I was able to reach **GitLab** on the *gitlab.test-gitpod.com*. **GitLab** asked for credentials where the username was **root**, and the password could be fetched from **Kubernetes** secret by 5.3.2. This password is available for 24 hours, so the recommendation is to change it for the **root** as soon as possible.

```
kubectl get secret \
  gitlab-gitlab-initial-root-password \
  -ojsonpath='{.data.password}' | base64 \
  --decode ; echo
```

I tested **GitLab** by pushing the **Pet Clinic** repo into it, and everything worked as expected.

5.3.3 Gitpod

I swapped **kubectl** to point into the **Gitpod** cluster and installed **Gitpod** by steps from 4.4. I also set up port forwarding on the router for port **2443**. After some time, all pods were in a **RUNNING** state except *ws-daemon* pod. I looked into the logs of this pod, and I found out *disable-kube-health-monitor* container using **bash** script, but the **k3s** image, running the cluster does not contain **bash**. According to the *discussion[32]*, **k3d** uses **k3s** image, not built on their side, but from Dockerfile[3]. So **k3d** does not provide a way to modify **k3s** image. However, there is a way to specify which image **k3d** will use for creating clusters. The comment[33] in the previous discussion is how it is possible to solve this problem. The solution was to build up my own image based on the **k3s** image and install bash here. I customized provided Dockerfile to meet my requirements, and I created my **k3s** image from 5.3.3 by *docker build . -t k3s*.

[3]https://github.com/k3s-io/k3s/blob/master/package/Dockerfile

```
FROM rancher/k3s:latest AS k3s

FROM alpine:latest
COPY --from=k3s / /
RUN apk add --no-cache bash curl nano
## This is as per-the parent image
RUN chmod 1777 /tmp
VOLUME /var/lib/kubelet
VOLUME /var/lib/rancher/k3s
VOLUME /var/lib/cni
VOLUME /var/log
ENV PATH="$PATH:/bin/aux"
ENV CRI_CONFIG_FILE="/var/lib/rancher\
        /k3s/agent/etc/crictl.yaml"
ENTRYPOINT ["/bin/k3s"]
CMD ["server", "--disable=traefik"]
```

Consequently, I had to delete the **Gitpod** cluster and create a new one with the just created image. I used command[5.3.3], which differs from the previous installation by the flag *-i k3s*, and it means **k3d** will use a customized image instead of the default **k3s** one. After the new installation of **Gitpod** *ws-daemon* pod was still failing to start because it requires */sys/fs/cgroup* and */proc* mounts to be shared. When I made these mounts shared it finally got into a **RUNNING** state.

```
sudo k3d cluster create -i k3s \
    -p 2443:443@loadbalancer \
    -v /tmp/workspaces:/var/gitpod/workspaces:shared \
    --k3s-arg "--disable=traefik@server:0" \
    gitpod

# Make mounts shared
sudo docker exec k3d-gitpod-server-0 \
  mount --make-shared /sys/fs/cgroup

sudo docker exec k3d-gitpod-server-0 \
  mount --make-shared /proc
```

When I hit *gitpod.test-gitpod.com* I was able to reach the self-hosted **Gitpod** dashboard. I tested the connection with **GitLab**, which I deployed in another cluster, by creating **OAuth App**[4] in **GitLab** and providing *client ID* and *client secret* to the **Gitpod**. There is one thing that does not work, and it is **Projects**. After creating a new project it is stuck in a loading state. I found the question[34] on the official **Gitpod** side, and it seems it is the problem on their side. It is relatively new, so I hope they solve it soon.

[4]https://docs.github.com/en/developers/apps/building-oauth-apps

5.4 One-click deployment

At this point, when I was able to deploy **Gitpod** and **GitLab** using **k3d**, I started working on the automation of the process. The goal was to provide an easy and quick solution to make **Self-Hosted Gitpod** and **Self-managed GitLab** work on a single machine. The first step was to find some **Infrastructure as Code (IaC)[35]** tool which supports the creation of **k3d** clusters[5.3] and Docker containers for the reverse proxy[5.3.1]. There are a lot of **IaC** tools like **AWS CloudFormation**[5], **Azure Resource Manager**[6], or **Google Cloud Deployment Manager**[7], but these tools are locked on one cloud provider. Some tools like **Terraform[38]** or **Ansible[39]** can work with multiple cloud providers. At a very high level, given the capabilities of both the products, **Terraform** and **Ansible** come across as similar tools. Both of them are capable of provisioning the new cloud infrastructure and configuring the same with required application components. The **Terraform** works best with orchestration, and **Ansible** is great at configuration management[37]. Since I needed a tool for creating infrastructure for **Gitpod** and **GitLab**, I chose the **Terraform**.

Terraform. **HashiCorp Terraform[38]** is the most popular and open-source tool for infrastructure automation. It helps in configuring, provisioning, and managing the infrastructure as code. With **Terraform**, is possible to easily plan and create **IaC** across multiple infrastructure providers with the same workflow. It uses the declarative approach to define the required infrastructure as code[36]. **Terraform** relies on plugins called **providers** to interact with cloud providers and other APIs. **Terraform** configurations must declare which providers it will use so that **Terraform** can install and use them. Additionally, some providers require configuration before they use it. Each provider provides sets of resources types and data sources that **Terraform** can manage. **Terraform Registry[41]** is a major directory of publicly available **Terraform** providers and hosts providers for most major infrastructure platforms[40].

I used *pvotal-tech/k3d*[8] provider for creating **k3d** clusters[5.3] and *kreuzwerker/docker*[9] provider for producting the reverse proxy[5.3.1] container. The configuration with **Terraform** was elementary. **K3d provider** allows me to specify **k3s** image, mount volumes to the cluster, specify ports and disable **Traefik**. **Docker provider** is also very intuitive, and it provides options to create the same reverse proxy server as in 5.3.1, and it is possible to build up docker images. I used this feature to create customized **k3s** images[5.3.3] during the creation of infrastructure[5.4].

[5]https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html
[6]https://docs.microsoft.com/en-us/azure/azure-resource-manager/
[7]https://cloud.google.com/deployment-manager/docs
[8]https://registry.terraform.io/providers/pvotal-tech/k3d/latest
[9]https://registry.terraform.io/providers/kreuzwerker/docker/latest

```
resource "docker_image" "k3s" {
  name          = "k3s"
  keep_locally  = false
  build {
    path = "k3s"
  }
}
```

Terraform needs to be initialized first by *terraform init*, and then it builds the whole infrastructure by command *terraform apply* and provided configuration. At this point, I could install the **Gitpod** and the **GitLab** on created infrastructure. It was similar to 5.3, but my goal was to automate it as much as possible.

5.4.1 Installation script

I created installation scripts for both applications where I defined steps to install the **Gitpod**, respectively the **GitLab**. I made one more script[5.4.1] where I ran **Terraform** followed by installation scripts for both applications.

```
terraform apply -auto-approve
./gitpod/install.bash
./gitlab/install.bash
```

This solution depended on the platform because the **docker provider** requires a docker host string in configuration. And when I ran it on **WSL** creation of workspace failed on a **Unix-related** error. When I ran it from **Windows power shell** installation scripts also failed because they are written in the **bash**. The solution was not very good, and it also felt weird to me, so I started thinking about another way.

5.4.2 Terraform apply

At this point, I was looking for a way to run scripts directly from **Terraform**. I found a feature called **Provisioners** used to model specific actions on the local machine or a remote machine to prepare servers or other infrastructure objects for service[42]. There are three types of basic provisioners:

- **file** - used to copy files or directories from the machine executing Terraform to the newly created resource
- **local-exec** - invokes a local executable after a resource is created
- **remote-exec** - invokes a script on a remote resource after it is created

The biggest issue I discovered is that most provisioners require access to the remote resource via **SSH**[10] or **WinRM**[11], and in my case, it would

[10]https://www.ssh.com/
[11]https://docs.microsoft.com/en-us/windows/win32/winrm/portal

require additional unnecessary configuration. I decided to use a simpler way. Since my solution uses **Docker** I used volume attachment instead of **file provisioner**. The **remote-exec provisioner** also requires connection to the resource, so what was left was the **local-exec provisioner**. There is an issue[43] where people ask about a **docker-exec provisioner**. But it is still open, so maybe there will be such a provisioner in the future.

I attached configuration files and the installation script to each container, and using a **local-exec provisioner** to ran the installation with *docker exec...* command. I ran into a few problems according to the install scripts. Since **GitLab** was installed with package manager **Helm** the installation failed because the docker image did not contain it. I had to update my **k3s** docker image to be able to use tools like **curl**, **openssl**, **tar**, and **helm**. **Gitpod** uses an installer, and it was also missing in the image, so I installed it in the installation script. There I ran into another issue with the **cert-manager**. The script was quick enough to run the installation of **Gitpod**, but the **cert-manager** was not ready yet, so the installation failed. I implemented a loop[5.4.2] checking if **cert-manager** pods are in the **RUNNING** state.

```
while [[ $(kubectl get pods -n cert-manager | \
          grep Running) == 3 ]]; do
  printf .
  sleep 10
done
```

It was a naive solution because sometimes the state of pods was **RUNNING**, but pods were not ready yet, and the installation failed anyway. So I had to implement something smarter and after some investigation, I found the command *kubectl rollout status deployment <deploymentName>*. The command requires the name of the deployment to check. Therefore I implemented a function[5.4.2] that fetches the names of all deployments in the specified namespace and runs the status command in the loop. I added the parameter to pass namespace, so I could reuse this function also for checking if the installation was done.

```
# Check if all deployments are ready
function check_deployments() {
    deployments="$(kubectl get deployments -n $1 \
                -o custom-columns=":metadata.name")"
    for deployment in $deployments; do
        kubectl rollout status deployment -n $1 $deployment
    done
}
# Function call
check_deployments "cert-manager"
```

At this point, I was able to start **Gitpod**, **GitLab**, and the **reverse proxy server** by one *terraform apply* command. I cleaned up the **Terraform** main file with variables and the configuration file. I also wanted to get rid of the hardcoded domain name in configuration files, so I passed it from the **Terraform** as an environment variable. This also required changes in installation scripts where I used the *sed*[12] command to replace all occurrences of a string *<domain>* with domain from the environment variable. The domain was hardcoded also in the reverse proxy configuration file, which passed as volume directly into the proxy configuration directory, and the proxy server used this configuration. Therefore I used a slightly different approach. I attached the volume with a configuration file with string *<domain>* instead of the domain name into the container. Then I used the *sed* command and redirected the output into the proxy configuration file, and finally, I had to reload the **Nginx proxy** config by *nginx -s reload*.

5.4.3 Auth Provider

Gitpod is connecting to a Git provider. It happens via the dashboard on the first launch or by providing **authProviders** configuration as a **Kubernetes secret**. In my case, I went for the second option to set up a provider during the installation. The configuration requires two secret strings, one for *client id* and another for *client secret*. For security reasons, *client id* and *client secret* cannot be hardcoded in the configuration. So I found another **Terraform** provider, *hashicorp/random*[13], which supports the use of randomness within **Terraform** configurations. I generated two random strings with a length of 64 characters and passed them into clusters as environment variables. Then I used **sed** to replace all occurrences of *<clientId>* and *<clientSecret>* with values from environment variables.

I started with **GitLab**, where the **PostgreSQL**[14] database needs to contain an **OAuth**[44] configuration. I created the SQL script[5.4.3], which was supposed to add an **OAuth** record into the database.

```
INSERT INTO oauth_applications (name, uid, secret,
                                redirect_uri, scopes,
                                created_at, updated_at,
                                owner_id, owner_type)
VALUES (
       'Gitpod',
       '<clientId>',
       '<clientSecret>',
       'https://gitpod.<domain>/auth/gitlab/callback',
       'api read_user read_repository',
       now(), now(), 1, 'User'
       );
```

Insertion to the database requires a password, which I fetched from **Kubernetes secret** from the **GitLab** cluster, and then I used *sed* to replace variables. Finally, I created a record in the database by the following commands[5.4.3].

[12]https://linux.die.net/man/1/sed
[13]https://registry.terraform.io/providers/hashicorp/random/latest
[14]https://www.postgresql.org/

```
# Fetch PostgresSQL password
DBPASSWD=$(kubectl get secret gitlab-postgresql-password \
  -o jsonpath='{.data.postgresql-postgres-password}' \
   | base64 --decode)

# Replace variables
SQL=$(sed "s+<clientId>+${CLIENT_ID}+g;
           s+<clientSecret>+${CLIENT_SECRET}+g;
           s+<domain>+${DOMAIN}+g" \
           "gitlab/insertOauthApplication.sql")

# Insert record in database
kubectl exec -it gitlab-postgresql-0 -- bash \
    -c "PGPASSWORD=$DBPASSWD psql -U postgres
    -d gitlabhq_production -c \"$SQL\""
```

For the **Gitpod**, I had to add an **authProvider** record[5.4.3] into the *gipod.config.yaml* file[4.4] and generate a secret with appropriate values[5.4.3]. After this configuration, I added commands[5.4.3] into the installation script for the **Gitpod**. The first command replaces secrets and the domain with values from the environment variables, and another one creates **Kubernetes secret**.

```
authProviders:
  - kind: secret
    name: public-github
```

```
id: Local GitLab
host: gitlab.<domain>
protocol: https
type: GitLab
oauth:
  clientId: <clientId>
  clientSecret: <clientSecret>
  callBackUrl: https://gitpod.<domain>/auth/gitlab/callback
  settingsUrl: gitlab.<domain>/profile/applications
```

```
# Replace variables
PROVIDER=$(sed "s+<clientId>+${CLIENT_ID}+g;
                s+<clientSecret>+${CLIENT_SECRET}+g;
                s+<domain>+${DOMAIN}+" \
                "gitpod/gitlab-oauth.yaml")

# Create secret
kubectl create secret generic \
      --from-literal=provider="$PROVIDER" gitlab-oauth
```

Chapter 6
Solved issues

In this chapter, I will mention some issues, I ran into during working with **Gitpod** deployed on my machine. I found solutions or workarounds for all of them, and in my opinion, it expanded my knowledge of **Gitpod**.

6.1 Missing git context

After I created a new workspace and I wanted to commit some change, the workspace complained about missing *user.name* and *user.email*. Moreover, when I set these values manually the commit worked, but the workspace asked for git credentials to push the change. This behavior puts on the future user some unpleasant work, which should be automatic. Furthermore, according to the official **Gitpod** documentation[3], it should be done by **Gitpod**. Therefore, I asked about this missing behavior in the **Gitpod** community[45]. The answer was to use some mechanisms provided by **Gitpod** to add git context into the workspace. The first suggestion was to set the remote URLs to include the username and the git personal data. I did not try this because I think sending personal tokens in the URL is not a very secure way of providing git context. Another suggestion was to use **dotfiles**. **Dotfiles** are a way to customize the development environment according to personal needs[46]. The idea is to store files in another repository. These files are downloaded into the workspace during the start. Furthermore, **Gitpod** recognizes scripts with names like *install.sh*, *bootstrap.sh*, or *setup.sh* among the files, and runs it. This approach with **dotfiles** requires storing the git personal access token in some repository, which is also not so much secure. So I started digging into it more, and I found that the workspace was missing a */home/gitpod/.gitconfig* file. The purpose of this file is to configure what I need. I created a workspace in the official **Gitpod**[https://gitpod.io/projects] and checked if the file exists here. The file[6.1] contains the name, email, and also credentials part. **Gitpod credential-helper** sets the credentials during the start of the workspace by *helper = /usr/bin/gp credential-helper*.

```
[push]
        default = simple
[credential]
        helper = /usr/bin/gp credential-helper
[user]
        name = stanislavlas
        email = stanislav.las@gmail.com
```

I created a file */home/gitpod/.gitconfig* in my self-hosted workspace and copied the file content[6.1] into it. It solved the missing context issue. So I was not asked for *user.name*, *user.email*, or *git credentials* anymore. At this point, I could use **dotfiles**[**46**] because there are no sensitive data in the *.gitconfig* file. After setting up a **dotfiles** repository and the URL, I ran into another issue when I initialized a new workspace Paradoxically, it could not download data from the repository because git context was missing in the workspace. I came up with a solution to this issue. I could store the file *.gitconfig* in the application repository and copy it into the correct location[6.1] by creating a workspace in the **before task**[**2.1.2**]. Finally, this approach worked as expected, and it creates a *.gitconfig* file during every creation of a workspace.

```
# Before task
- before: sudo sh ./.gitpod/setup.sh \
  $GITPOD_GIT_USER_NAME $GITPOD_GIT_USER_EMAIL

# Replace variables with actual values
sed "s+<user.name>+$1+g; s+<user.email>+$2+g" \
    ".gitpod/.gitconfig" > /home/gitpod/.gitconfig
```

6.2 Git-crypt

There must be some configuration file with appropriate URLs to communicate with other already deployed services. The file can store in the repository, but it needs some mechanism to be securely encrypted. **Git-crypt**[**47**] is a tool that allows encrypting and decrypting files with **GPG keys**[1] in the background of git commands. The files are encrypted on push to and decrypted when fetched from the remote server. The configuration consists of an installation with *apt-get install -y git-crypt* and the creation of the secret key for decryption. Sensitive files, which need encryption, can be defined in a *.gitattributes* file. I created a dummy secret file *api.key* for testing purposes and set up encryption for it[6.2].

```
api.key filter=git-crypt diff=git-crypt
```

The secret key is generated by *git-crypt export-key <path to key>*. Once data is encrypted by the secret key, it is decrypted only with the same key.

[1]https://www.privex.io/articles/what-is-gpg

Gitpod provides the way to define a custom dockerfile on which workspace will be running[48], so I created one with the installation of **git-crypt** and added it to the *.gitpod.yaml*[6.2].

```
FROM gitpod/workspace-full

RUN sudo apt-get update && \
    sudo apt-get install -y git-crypt
```

When I started a new workspace the creation crashed on error: ERROR: failed to mount /tmp/containerd-mount. Unfortunately, I could not solve this issue on my own, so I asked for help from the **Gitpod** community[49]. The suggestion was to use the **Ubuntu** image instead of the **Alpine** image.

I wanted to use a similar approach as from 5.3.3, but it did not work because the it copies the whole root directory from the **k3s** image and rewrites everything from **Ubuntu**. Therefore, I had to install all tools on the **Ubuntu** image from scratch[6.2].

```
FROM ubuntu:latest

RUN apt-get update && \
    apt-get install -y curl containerd

RUN curl -LO https://github.com/k3s-io/k3s/releases/\
    download/v1.23.5-rc5%2Bk3s1/k3s && \
    chmod 755 k3s && mv k3s /bin/

RUN curl -LO "https://dl.k8s.io/release/$(curl -L -s \
    https://dl.k8s.io/release/stable.txt)\
                        /bin/linux/amd64/kubectl" && \
    install -o root -g root -m 0755 kubectl \
    /usr/local/bin/kubectl && rm kubectl

RUN curl -fsSLO https://github.com/gitpod-io/gitpod/releases\
    /latest/download/gitpod-installer-linux-amd64 && \
    install -o root -g root gitpod-installer-linux-amd64 \
    /usr/local/bin/gitpod-installer

RUN chmod 1777 /tmp
VOLUME /var/lib/kubelet
VOLUME /var/lib/rancher/k3s
VOLUME /var/lib/cni
VOLUME /var/log
ENV PATH="$PATH:/bin/aux"
ENV CRI_CONFIG_FILE=/var/lib/rancher/k3s/agent/etc/crictl.yaml
ENTRYPOINT ["k3s"]
CMD ["server"]
```

After I made this work, I started a workspace with the **Ubuntu** image, but sadly, I ran into the same error as before. I tried one more thing I used *fsShiftMethod* instead of *fuse* in the **Gitpod** config file[4.4]. At this point, the *ws-daemon* pod failed with the error: Your kernel headers for kernel 5.4.72-microsoft-standard-WSL2 cannot be found.. That gives me an idea to run the whole infrastructure on **Ubuntu** instead of **Windows**. And finally, it solves the issue with dockerfile and the problem with missing git context[6.1].

I had to solve how to get **git-crypt-key** into the workspace. I built up a new repository for testing purposes, where I added the key. It is not a secure way, but I need to test if the key can fetch during the creation of the workspace. To make it secure, I would use some tools dedicated to storing sensitive data, for example, **Vault**[**50**].

I had the key in another place, so I added a new line into the workspace dockerfile[6.2], which is supposed to download the key in the image build.

```
FROM gitpod/workspace-full

RUN sudo apt-get update && \
    sudo apt-get install -y git-crypt

RUN wget https://github.com/stanislavlas/dotfiles\
    /blob/main/git-crypt-key?raw=true -O git-crypt-key
```

The last thing I had to add was *git-crypt unlock ~/git-crypt-key* into **before task**[**2.1.2**] to initialize **git-crypt** with the key. In conclusion, I could make **git-crypt** work automatically in the **Gitpod** workspace.

6.3 Extensions

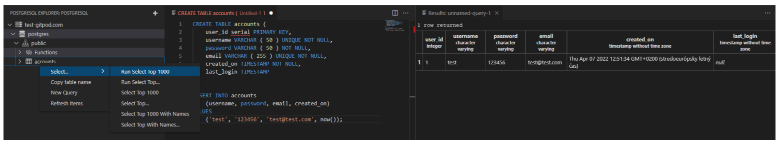
Gitpod allows the usage of **VS code** extensions in the workspace. Additionally, the definition of extensions is in the *.gitpod.yaml* file for every repository. It allows the owner to show which services might be used in the application. I tried to configure one of the extensions on my own. There are a variety of extensions for different tools like *Redis*, *SQL*, *Kafka*, and more[51]. I chose the **PostgreSQL Management Tool**[**52**], which is a query tool for **PostgreSQL databases**. Moreover, it provides database explorer as a visual aid to craft queries. It is necessary to include the extension[6.3] part in the *.gitpod.yaml* for proper installation into the workspace.

```
vscode:
  extensions:
    - ckolkman.vscode-postgres
```

Once I had the extension installed, I wanted to make some simple test if it works from the **Gitpod** workspace. I initialized a docker container[6.3] running the **PostgreSQL database**, where I defined the password and forwarded port. I also needed to allow the port on my router[4.1] to reach the database.

```
docker run --name postgres -d \
    -e POSTGRES_PASSWORD=123456789 \
    -p 5432:5432 \
    postgres
```

Consequently, I created a new workspace and opened the extension, in which I could add a new connection to the database. It asked me for connection details like the database hostname, username, password, and database name. At this point, I verified the established connection. In the explorer appeared a new item with a database name. The database was empty. Therefore I run scripts for the creation and fulfillment of a table[6.1].



```
CREATE TABLE accounts (
    user_id serial PRIMARY KEY,
    username VARCHAR ( 50 ) UNIQUE NOT NULL,
    password VARCHAR ( 50 ) NOT NULL,
    email VARCHAR ( 255 ) UNIQUE NOT NULL,
    created_on TIMESTAMP NOT NULL,
    last_login TIMESTAMP

SERT INTO accounts
    (username, password, email, created_on)
LUES
    ('test', '123456', 'test@test.com', now());
```

```
1 row returned
```

	user_id integer	username character varying	password character varying	email character varying	created_on timestamp without time zone	last_login timestamp without time zone
1	1	test	123456	test@test.com	Thu Apr 07 2022 12:51:34 GMT+0200 (stredoeurópsky letný čas)	null

Figure 6.1: PostgresSql extension

Unfortunately, I did not find a way to configure the connection automatically during the initialization of the workspace. The user will need to configure the appropriate extension after initialization. For facilitation, there can be some configuration files that will contain a list of relevant services and connection details to them. As I mentioned in the previous section[6.2], the files could be securely encrypted.

Chapter 7
Operability

7.1 Scalability

I did some benchmark tests to see how much I could get from **Gitpod** on my personal computer. **Gitpod** recommends at minimum two CPUs and 8GB of memory, but for a better experience is recommended to use at least a machine with four CPUs and 16GB of memory for node. The computer I ran benchmarks on has 16GB of memory and 8 CPUs, so it met into recommendations. I used deployment without **GitLab** to have the better overview of **Gitpod** itself. I was able to run four parallel workspaces on this setup. When I tried to create a new workspace, it got stuck in the creating state until some of the running workspaces timed out. The utilization of the memory and processor was not significantly high as we can see in the figure[7.1].

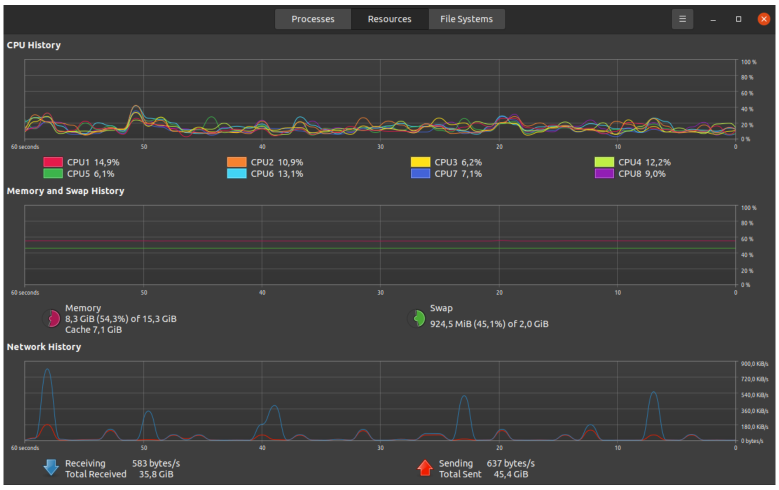


Figure 7.1: Utilization of four parallel workspaces

Since utilization pointed out that the computer can handle more, I thought I missed some restrictive configurations. I posted the question in the **Gitpod** community channel[53] and got the response that there might be a problem with pods. When I listed all **Gitpod** pods, it turned out that one of the pods was in the PENDING state. I checked pod logs and the reason for failure was 0/1 nodes are available: 1 Insufficient memory.

The reason why the utilization showed there were available resources and the pod still failed is **Kubernetes Resource Management for Pods and Containers**[54]. I described how to install **Gitpod** using the installer and the configuration file *gitpod.config.yaml* in the previous section[4.4]. The file contains the part which describes resource management for workspaces and defines that the workspace requests **one CPU and 2GB of memory**. It means the **kubelet**[1] reserves the requested amount of that system resource specifically for that container. The **Gitpod** requires a minimum of 8GB of memory, so when I created four workspaces, it reserved another 8GB. Therefore the computer was running out of memory, and I could not create a new workspace.

I lower the requested values to **the half of the CPU and 1GB of the memory** for the next benchmark. At this point, I could create nine parallel workspaces. We can see that the utilization of memory was similar to first benchmark, but the CPU is utilized a bit more in the 7.2.

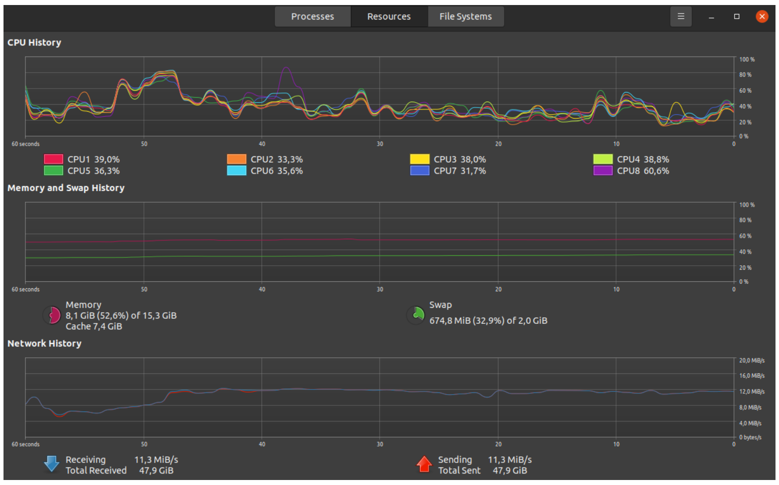


Figure 7.2: Utilization of nine parallel workspaces

[1]https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

I used **a quarter of the CPU and 500MB of the memory** in the last benchmark. The maximum number of concurrent workspaces I could run on my computer was nineteen. The utilization went rapidly up[7.3] and it consumed almost all the resources on the machine.

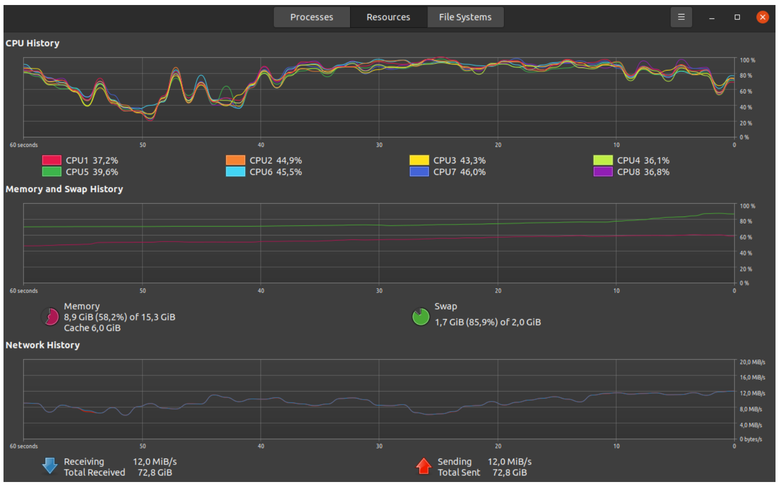


Figure 7.3: Utilization of nineteen parallel workspaces

7.2 Observability

The last part which was missing from reasonable usage was observability. **Observability** is the ability to measure a system's current state based on the data it generates, such as logs, metrics, and traces[55]. It could be achieved by the integration of **Prometheus[56]** on a **Kubernetes cluster**.

7.2.1 Prometheus

Prometheus is an open-source monitoring framework[57]. It provides out-of-the-box monitoring capabilities for the Kubernetes container orchestration platform.

- **Metric Collection** - **Prometheus** uses the pull model to retrieve metrics over HTTP.
- **Metric Endpoint** - The systems should expose the metrics on an */metrics* endpoint.
- **PromQL** - **Prometheus** comes with the language **PromQL** that can be used to query the metrics in the **Prometheus** dashboard.
- **Prometheus Exporters** - Exporters are libraries that convert existing metrics from third-party apps to **Prometheus** metrics format.
- **TSDB (time-series database)** - **Prometheus** uses TSDB for storing all the data efficiently.

The **Prometheus** monitoring stack for **Kubernetes** consists of three components:

- **Prometheus Server**
- **Alert Manager**
- **Grafana**

The following figure[7.4] shows how the components are connected.

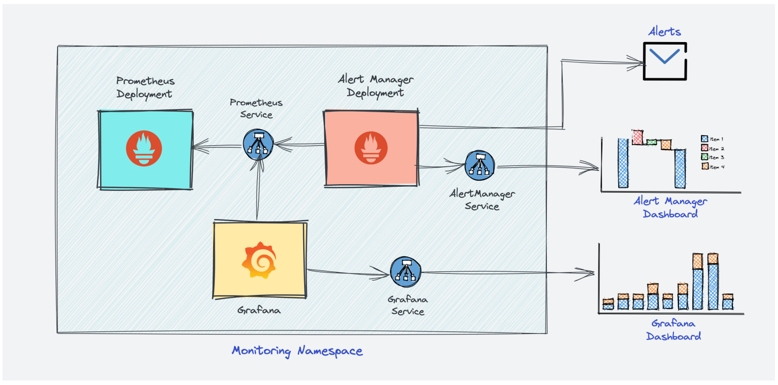


Figure 7.4: Kubernetes Prometheus monitoring stack[57]

For **Prometheus** implementation alongside **Gitpod**, I followed the instructions from the example[57]. First, I created a new namespace for observability tools. **Prometheus** uses **Kubernetes** API to read the metrics, so according to the guide, I defined a policy for the API and bound it to the namespace. The **Prometheus** configuration defines in the file *prometheus.yaml*, and rules for **Alert manager[7.2.3]** are in the file *prometheus.rules*. These files mount by **config map** to the **Prometheus** container in */etc/prometheus* location. The advantage of this approach is that when the configuration needs an update, it is enough to update the config map and restart the **Prometheus** pod. Next, the **Prometheus** deployment is triggered. It uses the latest **Prometheus** image from the docker hub. Finally, it exposes as a **Kubernetes** service on the port *30000*. After these steps, I could access the **Prometheus** dashboard on *localhost:30000*.

7.2.2 Kube state metrics

Kube state metrics primarily produce metrics in **Prometheus** format with the stability as the **Kubernetes** API. It also provides **Kubernetes** objects and resources metrics that cannot fetch directly from native **Kubernetes** monitoring components[58]. I used the example[58] to implement it into the cluster. **Kube state metrics** are available as a public docker image, and it requires deploying:

- **Service account**
- **Cluster role** for permissions to **Kubernetes** API

- **Cluster role binding**, which binds the service account with the cluster role
- **Service** to expos metrics

7.2.3 Alert Manager

Alert Manager[59] is an open-source alerting system that works with the **Prometheus** monitoring system. I followed the guide[59] to implement **Alert Manager** on **Kubernetes**. It requires deploying a config map for **Alert Manager** configuration. The configuration from the instructions uses an email and **Slack webhook** receivers. It also requires an alert template, also deployed with a config map. **Alert Manager** image is publicly available, and the image using for the deployment. Finally, the service exposes the **Alert Manager** endpoint was available on the *localhost:31000*. I created a simple rule to alert me when the container memory usage is high. After the deployment of the alert, the alert did not fire up because I did not configure receivers properly. It requires an **SMTP** server for the email receiver, which I did not have, so I decided to demonstrate the rule by **Slack webhook**[2]. I followed instructions to create a webhook, and it provides an URL, which I used in the **Alert Manager** configuration. When I initialized a new workspace memory, usage in the cluster raised, and the alert started firing[7.5].

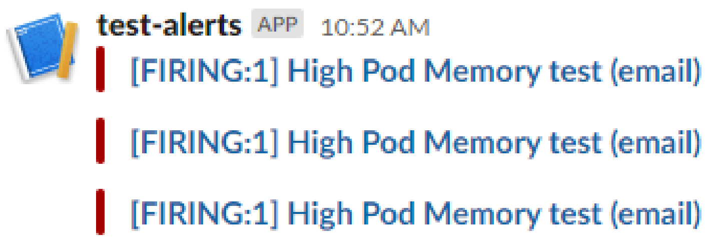

test-alerts APP 10:52 AM
[FIRING:1] High Pod Memory test (email)
[FIRING:1] High Pod Memory test (email)
[FIRING:1] High Pod Memory test (email)

Figure 7.5: High Pod Memory alert

7.2.4 Grafana

Grafana is an open-source lightweight dashboard tool. For the **Grafana** integration I used instructions from the guide[60]. Deployment requires configuration with the **Prometheus** endpoint, deployed as a config map. For the **Grafana** deployment is used official **Grafana** docker image. And in the end, it requires service, which exposes the **Grafana** to *localhost:32000*. The initial credentials are username: *admin*, password: *admin*, and it prompts to change the password after the first login. There are many community dashboard templates available for **Kubernetes**. Therefore it is easy and

[2]https://api.slack.com/messaging/webhooks

quick to import a prebuild dashboard for **Prometheus** metrics. The detailed instructions on how to import a prebuild dashboard are in the guide[60].

At this point, observability worked after manual setup, so I decided to automate it alongside the installation of **GitLab** and **Gitpod** from the previous chapter[5.4]. So I downloaded all configurations for **Prometheus**[**57**], **Kube state metrics**[**58**], **Alert Manager**[**59**], and **Grafana**[**60**] into the */observability* folder in the **one-click deployment**[**5.4**] repository. After I had the configuration files ready, I created the script[7.2.4], which creates a *monitoring* namespace and runs all steps to deploy **Prometheus monitoring** into the cluster. It also checks if all monitoring tools got successfully deployed, and in the end, it prints URLs for observability tools.

```
#!/bin/bash

function check_deployments() {
    deployments="$(kubectl get deployments -n \
                    $1 -o custom-columns=":metadata.name")"
    for deployment in $deployments; do
        kubectl rollout status deployment -n $1 $deployment
    done
}

echo "Installing observability"

kubectl create namespace monitoring

kubectl create -f observability/kubernetes-prometheus/
kubectl apply -f observability/kube-state-metrics-configs/
kubectl create -f observability/kubernetes-alert-manager/
kubectl create -f observability/kubernetes-grafana/

check_deployments "monitoring"

echo "Premotheus endpoint: localhost:30000"
echo "Alert manager: localhost:31000"
echo "Grafana: localhost:32000"
```

Next, I needed to forward ports for the tools to be able to hit them from the outside of the container. I attached the */observability* folder to the container, and also I added the possibility to turn on and off observability installation. It required the passing of a new environment variable. In the **Gitpod** installation script, I added a condition that checks if observability is enabled[7.2.4].

```
if [ "$ENABLE_OBSERVABILITY" = "true" ]
then
    ./observability/install.bash
fi
```

At this point, the initialization of the observability is triggered by **Terraform** variable.

Chapter 8
Conclusion

The purpose of the thesis was to explore **Cloud-based Development**[**1**] and its possibilities for usage in the private **cloud**. Currently, two major platforms deal with it, **Gitpod**[**3**] and **GitHub Codespaces**[**7**]. What are the platforms offering? Basically, they provide the possibility to create a virtual development environment fulfilling a task in the **cloud** from the git context. The virtual workspace destroys itself after finishing the job. **Cloud-based Development** is a big step to boosting productivity when we think about constantly switching the task contexts or remote work. Both platforms provide an environment similar to a local computer with the **Linux** operating system. Additionally, environment configuration is stored as a code that allows reproducibility of the workspaces. The platforms use **VS Code** editor that is customizable and configurable on the team, project, or individual level. Security is enforced by the **HTTPS** connection to the remote workspace using **TLS** certificates. Moreover, the source codes are stored in the one place of trust on the **cloud**, which is accessible only via a secure connection of a web browser. **GitHub Codespaces** is currently available only for users with **GitHub Team** or **GitHub Enterprise** subscriptions. On the other hand, **Gitpod** is free to use for up to fifty hours per month for individuals. Furthermore, it provides a possibility to install **Gitpod Self-hosted**[**12**] on the customer‘s infrastructure.

In the thesis, I described the process of the **Gitpod** installation on different platforms using different tools. Finally, I created a guide for building the infrastructure and installing **Gitpod** and **GitLab** using **Terraform** on one machine, which I shared with the **Gitpod community**[**61**]. The community also helped me to solve issues with git context[45], errors with creating a workspace[49], and parallel workspaces issues[53]. I provided solutions for all the questions, so I believe that in case someone runs into similar problems, it will help. When I overcame these issues, every workspace I create can clone a git repository, and a user is able to commit and push changes without restrictions. Every workspace also allows installing **VS Code** extensions for connection to relevant services like databases. Unfortunately, I could not find a way to automatically configure extensions, but the connection strings can be stored in the configuration files alongside URLs to other services. The configuration files can be encrypted and decrypted in the background of git commands with **git-crypt**[47]. In this case, the connection strings do not expose in plaintext in the git repository. Eventually, I integrated observability on the cluster and did benchmarks, where I could run a maximum of nineteen parallel workspaces on my machine. There is a limitation from the **Self-hosted Gitpod** on the number of users using **Gitpod** without fees. The maximum number of users is ten, and after reaching the limit, every new user will cost 29€ per month. After creating the tenth user, **Gitpod** stopped responding to the new creation when I tested the behavior.

In conclusion, I think the **Gitpod** is a helpful feature when a company has enough resources for running its self-hosted. The limitation on the number of users is a disadvantage for smaller companies with dozens of users, which would not have the large budget for **Gitpod**. I think the management of **Gitpod** will be open to price negotiation in the case of corporates with a large number of users. Overall, in my opinion, **Cloud-based Development** is the direction that software development should take. The predefined environment configuration allows to simply run an application for the developer, manager, sale, or even customer. Remote access from various devices like computers, mobile phones, or tablets is also a practical convenience. I believe this thesis will aid people in better understanding **Cloud-based Development**, especially **Gitpod**. And I hope I use the knowledge I gained in this thesis in my future career, and maybe I will work on some interesting **Cloud-based** projects.

Bibliography

[1] Shkurhan Gregorii, *What Is Cloud Development? The fundamentals You Need To Know In 2021*, 30.09.2021, https://northell.design/blog/cloud-development-do-your-business-need-it/

[2] Friedman Nat [@natfriedman], *End of year. If you really need it before then you can buy a Team account*, 11.08.2021, https://twitter.com/natfriedman/status/1425508910476271624?s=20

[3] Gitpod, *Introduction to Gitpod*, 2021, https://www.gitpod.io/docs/

[4] GitHub Inc, *Contexts*, 2021, https://docs.github.com/en/actions/learn-github-actions/contexts

[5] Gitpod, *Start tasks for Prebuilds & New Workspace*), Start Tasks, 2022, https://www.gitpod.io/images/docs/beta/configure/start-tasks/prebuilds-new-workspace.png

[6] Gitpod, *Getting Started*), 2022, https://www.gitpod.io/docs/getting-started

[7] GitHub Inc, *GitHub Codespaces overview*, 2022, https://docs.github.com/en/codespaces/overview

[8] GitHub Inc, *Introduction to dev containers*, 2022, https://docs.github.com/en/codespaces/setting-up-your-project-for-codespaces/configuring-codespaces-for-your-project

[9] GitHub Inc, *GitHub Team*, 2022, https://docs.github.com/en/get-started/onboarding/getting-started-with-github-team

[10] GitHub Inc, *GitHub Enterprise Cloud*, 2022, https://docs.github.com/en/get-started/onboarding/getting-started-with-github-enterprise-cloud

[11] GitHub Inc, *Deep dive into Codespaces*, 2022, https://docs.github.com/en/codespaces/getting-started/deep-dive

[12] Gitpod, *Gitpod Self-Hosted*, 2022, https://www.gitpod.io/docs/self-hosted/latest

[13] Docker Inc, *Docker*, 2022, https://www.docker.com/

[14] Docker Inc, *Docker*, 2022, https://www.docker.com/products/docker-desktop

[15] Docker Inc, *Deploy on Kubernetes*, 2021, https://docs.docker.com/desktop/kubernetes/

[16] Microsoft, *Windows subsystem for linux*, 2022, https://docs.microsoft.com/en-us/windows/wsl/install

[17] The Kubernetes Authors, *Install Tools*, 2022, https://kubernetes.io/docs/tasks/tools/

[18] Helm Authors, *The package manager for Kubernetes*, 2022, https://helm.sh/

[19] Gitpod, *Troubleshooting Gitpod Self-Hosted*, 2022, https://www.gitpod.io/docs/self-hosted/latest/troubleshooting

[20] Canonical Ltd, *High availability K8s*, 2022, https://microk8s.io/

[21] Cornelius A. Ludmann [@corneliusludmann], *GitLab and Gitpod installation on k3d*, 03.07.2020, https://github.com/corneliusludmann/k3d-gitlab-gitpod

[22] containerd Authors, *Containerd*, 2022, https://containerd.io/

[23] Dabit Nader, *GitHub Codespaces vs Gitpod – Full Stack Development Moves to the Cloud*, 30.08.2021, https://www.freecodecamp.org/news/github-codespaces-vs-gitpod-cloud-based-dev-environments/

[24] Canonical Ltd., *Getting started*, 2022, https://snapcraft.io/docs/getting-started

[25] Gitpod, *Installation requirements for Gitpod Self-Hosted*, 2022, hthttps://www.gitpod.io/docs/self-hosted/latest/requirements

[26] Volker Thiel [@riker09], *Unable to disable Traefik*, 04.12.2019, https://github.com/k3s-io/k3s/issues/1160#issuecomment-561572618

[27] GitLab Inc, *Installing GitLab using Helm*, 2022, https://docs.gitlab.com/charts/installation/

[28] GitLab Inc, *Deployment Guide*, 2022, https://docs.gitlab.com/charts/installation/deployment.html

[29] Rancher, *K3s - Lightweight Kubernetes*, 2021, https://rancher.com/docs/k3s/latest/en/

[30] k3d Authors, *K3d*, 17.02.2022, https://k3d.io/v5.3.0/

[31] F5 Networks Inc, *What Is a Reverse Proxy Server?*, 2022, https://www.nginx.com/resources/glossary/reverse-proxy-server

[32] Simon Emms [@MrSimonEmms], *Can I install /bin/bash on the Docker image?*, 20.12.2021, https://github.com/k3d-io/k3d/discussions/901

[33] Simon Emms [@MrSimonEmms], *Can I install /bin/bash on the Docker image?*, 20.12.2021, https://github.com/k3d-io/k3d/discussions/903

[34] Max Alber [@MaxAlber], *Projects support for Gitpod Self-hosted Instances*, 02.03.2022, https://github.com/gitpod-io/gitpod/issues/8536

[35] Navdeep Singh Gill, *What is Infrastructure as Code? Best Practises / Benefits / Adoption*, 17.12.2020, https://www.nexastack.com/blog/infrastructure-as-code

[36] Navdeep Singh Gill, *Infrastructure as Code Tools to Boost Your Productivity in 2022*, 02.03.2022, https://www.nexastack.com/blog/best-iac-tools

[37] Sumeet Ninawe, *Terraform vs. Ansible : Key Differences and Comparison of Tools*, 01.10.2021, https://spacelift.io/blog/ansible-vs-terraform

[38] HashiCorp, *Terraform*, 2022, https://www.terraform.io/

[39] Red Hat, Inc, *Red Hat Ansible Automation Platform*, 2022, https://www.ansible.com/

[40] HashiCorp, *Providers*, 2022, https://www.terraform.io/language/providers

[41] HashiCorp, *Terraform registry*, 2022, https://registry.terraform.io/browse/providers

[42] HashiCorp, *Provisioners*, 2022, https://www.terraform.io/language/resources/provisioners/syntax

[43] Carlo Cabanilla [@Carlo Cabanilla], *Support provisioning using docker exec*, 15.01.2016, https://github.com/hashicorp/terraform/issues/4686

[44] Gitlab, *OAuth 2.0 identity provider API*, 2022, https://docs.gitlab.com/ee/api/oauth2.html

[45] Stanislav Las [@Stanislav Las], *Git context is missing*, 26.03.2022, https://discord.com/channels/816244985187008514/879915120510267412/957196500692267019

[46] Gitpod, *Dotfiles*, 2022, https://www.gitpod.io/docs/config-dotfiles

[47] Alexander Kus, *How to secure sensitive data with git-crypt*, 09.03.2020, https://buddy.works/guides/git-crypt

[48] Gitpod, *Custom Docker Image*, 2020, https://www.gitpod.io/docs/config-docker

[49] Stanislav Las [@Stanislav Las], *ERROR: failed to mount /tmp/containerd-mount*, 27.03.2022, https://discord.com/channels/816244985187008514/879915120510267412/957682548497080321

[50] HashiCorp, *Manage Secrets and Protect Sensitive Data*, 2022, https://www.vaultproject.io/

[51] Ilana Brudo, *Top 40+ VSCode Extensions for Developers in 2022*, 02.12.2021, https://www.tabnine.com/blog/top-vscode-extensions/

[52] Chris Kolkman, *PostgreSQL Management Tool*, 21.01.2022, https://marketplace.visualstudio.com/items?itemName=ckolkman.vscode-postgres

[53] Stanislav Las [@Stanislav Las], *Open 4 parallel workspaces*, 13.04.2022, https://discord.com/channels/816244985187008514/879915120510267412/96379750745075309

[54] The Kubernetes Authors, *Open 4 parallel workspaces*, 27.03.2022, https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

[55] Jay Livens, *What is observability?*, 01.10.2021, https://www.dynatrace.com/news/blog/what-is-observability-2/

[56] Prometheus Authors, *Prometheus*, 2022, https://prometheus.io/docs/introduction/overview/

[57] Bibin Wilson, *How to Setup Prometheus Monitoring On Kubernetes Cluster*, 28.01.2022, https://devopscube.com/setup-prometheus-monitoring-on-kubernetes/

[58] Bibin Wilson, *How To Setup Kube State Metrics on Kubernetes*, 27.01.2022, https://devopscube.com/setup-kube-state-metrics/

[59] Bibin Wilson, *Setting Up Alert Manager on Kubernetes – Beginners Guide*, 27.01.2022, https://devopscube.com/alert-manager-kubernetes-guide/

[60] Bibin Wilson, *How To Setup Grafana On Kubernetes*, 29.01.2022, https://devopscube.com/setup-grafana-kubernetes/

[61] Stanislav Las [@Stanislav Las], *Gitlab and Gitpod on k3d using Terraform*, 25.03.2022, https://discord.com/channels/816244985187008514/931200235601023016/9569048135206379