FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Application of containers in continuous software development
Student:	Bc. Henrich Le
Supervisor:	Ing. Petr Ciochoň
Study Programme:	Informatics
Study Branch:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	Until the end of winter semester 2021/22

Instructions

Analyze requirements for continuous integration and continuous delivery in the software-house environment provided by the supervisor.
Design and implement a proof-of-concept prototype of a CD/CI pipeline while satisfying the following requirements:

- use container-based virtualization,
- use serverless cloud computing execution model,
- use multitenant operational mode,
- provide self-serviceability within project teams,
- integrate with the existing identity and access management solution.

Design suitable metrics for assessment of the impact of the proposed solution on existing software project management and perform a basic evaluation of this impact.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 17, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Application of containers in continuous software development

Bc. Henrich Le

Department of Computer Systems
Supervisor: Ing. Petr Ciochoň

May 28, 2020

Acknowledgements

I would like to thank my supervisor, Petr Ciochoň, for technical direction and valuable insights during writing this thesis and Miriama Bernátová for consulting stylistic adjustments in text.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 28, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Le, Henrich. *Application of containers in continuous software development .* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Táto práca sa zameriava na návrh riešenia pre spúšťanie „pipeline" pre kontinuálnu integráciu a dodávku pre konkrétne vývojové prostredie s maximálnym využitím kontajnerov. Zaoberá sa opisom tohto prostredia, analýzou požiadaviek a pripravou metrík, ktoré sa neskôr využívajú na prieskum dostupných technológií. Medzi tieto technológie patrí *Jenkins*, *GitLab* a *Tekton*. Žiadna z týchto technológií nespĺňa všetky stanovené požiadavky, a preto je navrhnuté vlastné riešenie využívajúce mnoho princípov využívaných v skúmaných technológiách. Toto riešenie je navrhnuté, implementované ako dôkaz konceptu a otestované. Ponúka škálovateľný „serverless" model exekúcie, využívajúci technológiu orchestrácie kontajnerov *Docker Swarm* spolu s riadením prístupu integrovaným s externým poskytovateľom identít *OAuth2.0*. Taktiež je na navrhovanom riešení vykonané vyhodnotenie požiadaviek a metrík.

Klíčová slova kontajnerizácia, kontinuálna integrácia, kontinuálna dodávka, integrácia, pipeline, docker, Jenkins, GitLab, Tekton, Docker Swarm, autorizácia, autentifikácia, OpenID Connect, OAuth2.0

Abstract

This thesis is focused on designing solution for execution of continuous integration and delivery pipeline for specific development environment with maximum use of containers. It does so by describing this environment, analyzing requirements and preparing metrics which are then used for research of available technologies. These technologies include *Jenkins*, *GitLab* and *Tekton*. None of those technologies satisfy all the requirements and therefore custom solution utilizing many principles used in researched technologies is proposed. This solution is designed, implemented in proof-of-concept and tested. It offers scalable serverless execution model leveraging *Docker Swarm* container orchestration technology along with access control integrated with external *OAuth2.0* Identity provider. Also evaluation of requirements and metrics is performed on proposed solution.

Keywords containerization, continuous integration, continuous delivery, integration, pipeline, docker, Jenkins, GitLab, Tekton, Docker Swarm, authorization, authentication, OpenID Connect, OAuth2.0

Contents

List of Figures

List of Tables

Introduction

Creating software does not only contain tasks such as writing code and compiling it but it is getting increasingly complex with more and more tasks required. These tasks may include testing, accepting, releasing and many other. Relations between these tasks are getting progressively intricate and with this complexity comes a need to organize them into stages in which they are executed. Focus of a developer is fragmented between these tasks or stages and attention to creating software itself is significantly lowered. Demands on quality and security of a software are continuously rising. To supply these demands and to lower the time software spends in one of the stages, emphasis on accelerating software development cycle is required. To support this emphasis, need for automation, access control, separation and many other grows.

Usually more than one developer works on software, using resources required to execute tasks and stages previously mentioned. Therefore, it is crucial to have possibility to share these resource between individual software's tasks, together with separating them on access layer. In the past, this was achieved by using hardware servers. Later on by virtual machines on said servers. And now, thanks to Docker and its fundamental simplification and popularization of containerization, containers are one of the main means of sharing and separating resources simultaneously.

Aim of this thesis is to demonstrate proof-of-concept of a solution intended for continuous integration and delivery pipelines execution which will serve as a foundation for further implementation in specific software house environment. This will be achieved by creating appropriate metrics and requirements by which various technologies are evaluated.

At first, boundaries, in which this thesis will operate, must be set, this will be done by specifying software house environment, which will use results of this thesis. Next, basics of continuous software development and containerization will be described. Following this, requirements for solution will be constructed along metrics, based on those requirements. Metrics will serve for an evaluation of technologies described in consecutive chapter. After research

of technologies, design for such solution will be laid down with its implementation and basic testing in subsequent chapters. Last but not least, metrics created in earlier chapter will be used to evaluate solution proposed in previous chapters.

Chapter 1

Software house environment

This chapter describes software house environment since the goal of this thesis is to analyze requirements and to propose a solution for continuous integration and delivery for this environment.

Software house environment is an environment dedicated to building and maintaining software products.

The environment described in this chapter is derived from real software production organization with more than one thousand developers and hundreds of software projects.

1.1 Software development

Software in this environment is developed by small teams with about five members, using agile approach. This means changes to software occur often with regular releases. Therefore building, testing, packaging and releasing is done in frequent short cycles. This creates need for an uniform solution for continuous integration and delivery of software.

Teams are independent units, responsible for building, packaging and releasing software, as was mentioned above. Teams are not limited to one project and can work on multiple projects simultaneously.

Projects might have more than one team working on it at the same time. As not every project is equal, access to some projects is limited. Diverse projects for different customers exist in this environment. Hence isolation of teams is very important.

1.2 Technology

Most software produced in this environment is built upon custom web application framework, consisting of complete technology stack, ranging from client front end, server back end, storage up to container image deployed in

production. This framework is available in several programming languages, not in any particular order: Java, Node.js and C#. This helps to streamline development and establish standards in creating software. Usage of this framework also means that development teams do not have to experiment with every product. It provides a standard way of preparing local environment for a developer.

To store and version source code, **git** is used with git-flow branching model. Basic idea of this model is to have branch for each new feature, which is merged to develop branch. Develop branch serves as "staging ground" for changes before being integrated to master branch.[3]

Objects related to software development are uploaded to software repositories managed by **Nexus repository manager**, which is technology able to host, proxy and group multiple repository technologies such as *java's Maven artifacts*, *node's npms*, *docker's registry images*, arbitrary raw byte sequence (e.g. *zip*) and many more.

Identities for authorization are provided by *Open ID Connect protocol* built upon *OAuth2.0*

Released applications are packaged to containers, described more in chapter 3, and run with **Docker Swarm** container orchestration engine.

1.3 Current continuous integration and delivery

Large number of decentralized development teams work in this environment, divided by tree organizational structure. Therefore fairly large number of organizational units exists within it. Every unit has its own delivery team, which ensures that software is deliverable. This is done with help of local quality assurance team.

Due to this decentralization many teams started using variety of different tools for CI/CD, such as *TeamCity*, *Jenkins*, or no tools at all. These tools are in general non-flexible solution with static configuration and small scalability options, which usually provide no central authentication, nor authorization. Maintaining these tools usually requires additional team, which must take care of keeping them accessible, up-to-date and secure. This makes them more difficult to manage.

Chapter 2

Continuous software development

Continuous software development is an approach to software engineering, in which new versions of software are continually integrated, released and deployed. It consists of three main parts:

- continuous integration,
- continuous delivery,
- continuous deployment.

Each of these parts are closely related to each other and are dependent on the outputs of the previous one.

For an illustration, implementing a new version of software can be divided into various stages such as writing code, compilation, integration with main version of software, testing, packaging, accepting changes, release, deployment and maintenance. If process of executing them is done continuously, these stages belong to one of the parts of Continuous software development.

Relation between individual stages in context of continuous integration, delivery and deployment is shown in figure 2.1.

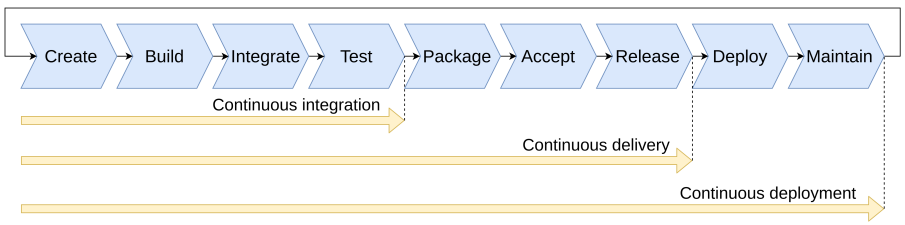

Figure 2.1: Relation between continuous integration, delivery and deployment

To appropriately explain these three parts, basics of *application deployment*, *version control system*, *software repository* and *quality assurance* are described.

2.1 Application deployment

Application deployment, also known as a software deployment is a process of planning, maintaining, and executing on delivery of a specific software version [4]. In other words, this process makes software available for users.

It can be made available for use in specific environment, which describes set of configuration, topology and computing resources used by a software. Environment can be dedicated, but is not limited to: *development*, *integration*, *staging* and *production*, where every environment has its purpose.

Development environment is used to develop software or a functionality by developer or small team. Isolation of environment enables frequent and unstable changes to occur without worrying about affecting other versions of software. **Integration environment** is where changes of development environments meet. Its goal is to combine work of developers or teams working on software and ensure certain level of quality of software before it is promoted to **staging environment**. This environment is used to simulate production as much as possible. It can be used for demonstrations, or to reveal breaking changes, which would not be possible to detect in any of previously mentioned environments. **Production environment** is dedicated to consumer use, hence most stable version of a software is here.

Node is used to represent resources available for a software in an environment . *Node* can represent physical hardware, virtual machine or container (more about containers in section 3) and has defined processing power, memory, storage and network.

2.2 Version control system

Version control system (VCS) is a system designed to manage software source code. It enables management of multiple versions of source code shared among developers. Every developer works on its own version of a software and all the changes made are then merged to main version. This by design supports distributed cooperation among developers.

As work on software is distributed, changes made to functionality can overlap. This can lead to conflicts among changes, usually referred to as merge conflicts. It is up to developer to resolve merge conflicts, and resolving them can take a significant amount of time, depending on the number of changes in each version.

One of the most popular technologies used for version control is *git*[5]. In git every version is referred to as a branch. These branches can ”branch out”

from other branches, but mainly from the main version of a code, usually labeled as "master" [6, Chapter: Git Branching]. This version of code is usually run on production environment.

2.3 Software repository

Software repository, or repository for short, is used to store packages and other immutable objects related to software during development. These objects are called artifacts and may represent dependencies required for software build, outputs of build process, or a distribution package of created software. It groups packages by its technology, e.g. java archives, node modules or ruby gems.

2.3.1 Repository manager

Repository manager is a server used to manage multiple types of repository. These include different repository technologies, usage types, such as hosted, proxy or group repository and repositories for different environments, e.g development, integration, staging and production.

One of the main purposes of repository manager is to provide single access control for different repositories, to control remote sources used in software and to keep track of artifacts produced by built systems [7].

2.4 Quality assurance

Quality of a software can be ensured by running various types of tests. *Unit test*, for example, is short block of code which calls method with predefined input and compares it to predefined result. Another common tests are *integration tests*, which verify if the individual components of a system work together. For example, if software integrates correctly with other software interface such as database, queues and other. After *integration tests* comes *system tests* which test system as a whole. This can include performance testing, reliability testing, scalability testing, security testing, and many other. *Acceptance tests* checks software functionality from user perspective. For example it tests whether record is created, saved to database, and loaded from database correctly [8]. Another means of ensuring quality of a software is nonfunctional testing, such as static code analysis, code coverage and so on.

Each of these tests are done in different stage of software development, for example unit tests, static analysis and code coverage are executed when software is built, integration and acceptance tests only after unit tests are passed and performance tests are executed after software's functionality was verified. There are numerous other types of tests but those are not discussed here, as testing is not goal of this thesis.

2.5 Continuous integration

Continuous integration (CI) is a process of integrating new code with predetermined quality to master branch as frequently as possible.

This differs from integrating branches in longer periods of time, such as weeks, or months as number of changes in code are much bigger, thus finding and isolating breaking change is much more difficult. On the other hand, with frequent integration of smaller change sets, finding change causing problems is much easier [4].

Quality assurance in continuous integration oftentimes include unit testing, static code analysis, code coverage and other. If build passes these tests, process of accepting this build as a deployable can start.

2.6 Continuous delivery

Continuous delivery (CD) is a process ensuring software can be deployed. This is achieved by using set of automation tools and principles that allow fast and regular software release [4]. Continuous delivery and integration are tightly coupled together as continuous delivery builds on outputs of continuous integration.

In continuous delivery quality is assured by executing acceptance and system tests.

Process of continuous delivery can be described by **delivery pipeline**. As Jez Humble and David Farley described in [9], delivery pipeline is an automated manifestation of process for getting software from version control to release. High level overview of delivery pipeline is shown in figure 2.2.

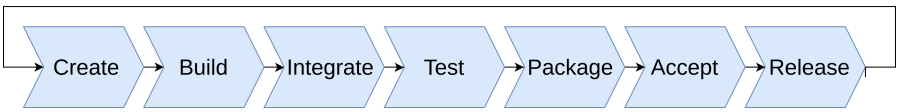

Figure 2.2: Example of a delivery pipeline

This manifestation has evolved over time to address continuous deployment as well. Therefore nowadays, if CI/CD pipeline is mentioned, it can refer to automated process of continuous integration, delivery and sometimes continuous deployment as well.

2.7 Continuous deployment

Continuous deployment is a process of deploying new versions of software to production environment as frequently as possible with minimal downtime.

In simpler pipelines continuous deployment is included a delivery process and more complex deployment make use of specialized tools such as Spinnaker [10].

This thesis main focus is proposing solution for continuous integration and delivery.

Chapter 3

Containerization and its orchestration

As was stated in section 2.6, delivery pipeline is an **automated** manifestation of delivery process. This process is executed over various projects built upon various technologies. Therefore it is necessary to use technologies which provide control over consumed resources and separation of individual environments. By using such technologies it is possible to use different environment configuration, different technologies, etc. side by side independently of each other. Containerization is technology which provides this with type of separation.

To describe containerization, one must start with virtualization. In a non-virtualized system, applications communicate via libraries to operating system and its kernel's interaction with hardware to execute operations. As more applications are running on same system, competition for system resources grows. There are also potential security risks as all applications share one file system and a process tree. This situation is illustrated in figure 3.1

3.1 Virtualization

Virtualization is a process of running virtual instance of computing system in a layer abstracted from actual hardware [11]. This ensures separation of individual systems for security and resource delegation benefits. Also it enables to emulate various types of hardware or other operating systems on top of existing hardware with existing operating system.

There are multiple ways to achieve this, for example, **full virtualization** on hardware layer using CPU assistance (Intel VT or AMD-V)[12]. Software responsible for this virtualization is called a hypervisor and virtualized computing system is known as a guest. This type of virtualization offers strong separation of guest operating systems from each other and hypervisor itself.

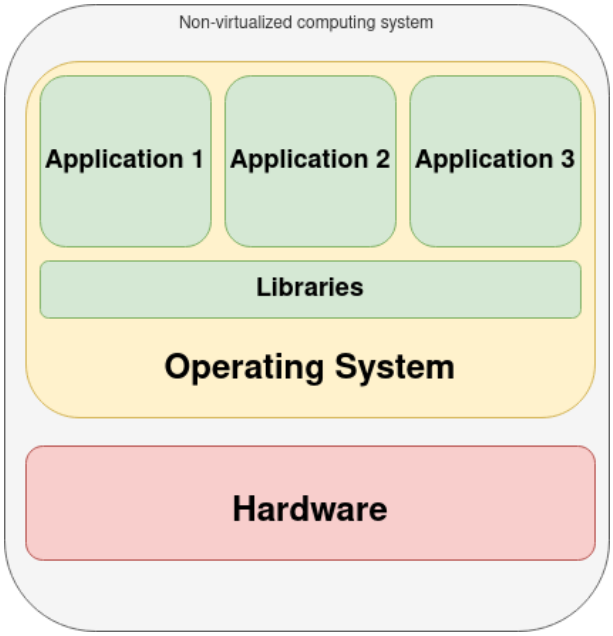

Figure 3.1: Non-virtualized system

Individual guests usually have its file system and process tree represented in hypervisor as a single file and process respectively. This is illustrated in figure 3.2.

Guest operating system's requests for hardware resources are translated by hypervisor to an actual interaction with hardware. This emulation might overload hypervisor, thus making it bottleneck of virtualization. Also space concerns might arise as every guest system needs its whole file system. This can contribute to slower start-up times ranging from one to more than ten minutes [13].

3.2 Containerization

Another way to virtualize computing system is to use OS-level virtualization, sometimes referred to as containerization.

Main idea is to partition hardware resources at operating system level making guests share a single kernel. This is in contrast to Full virtualization, which enables to emulate various types of hardware and run operating systems other than hypervisor's.

To achieve this virtualization, operating system kernel must allow multiple isolated name space groups and ideally resource sharing management. Comparison of full virtualization and OS-level virtualization is illustrated in figure

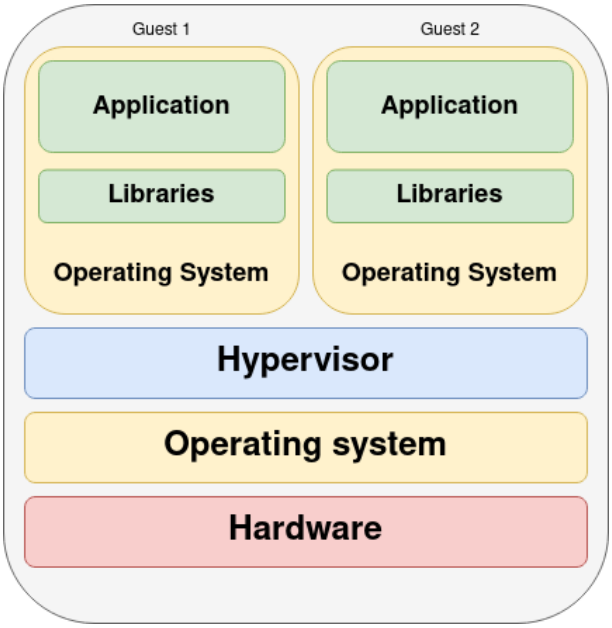

Figure 3.2: Full virtualization

3.3

Many technologies enable this type of virtualization, such as containers (Docker), zones (Solaris [14]) or jails (FreeBSD jail [15], unix chroot), and in recent years, docker containers has gained popularity, mainly due to its ease of use.

Container, according to Docker, is a standard unit of software that packages code and all its dependencies so the application runs quickly and reliably from one computing environment to another [16].

This means that applications can be developed, delivered and deployed in the same way. This eliminates many problems where environment of developer is different from production environment whether it is different version of library, different environment variables of a machine, etc.

There are many more containerization technologies besides docker (e.g. *podman*, *rkt*, *LXC*, *LXD*), but they will not be described further as it is not the main focus of this thesis.

Docker containerization technology was chosen for this thesis as it is widely used across many production environments, its large user community and its orchestration tool *Swarm* (described in section 3.5), which is used in provided software house environment, described in chapter 1.

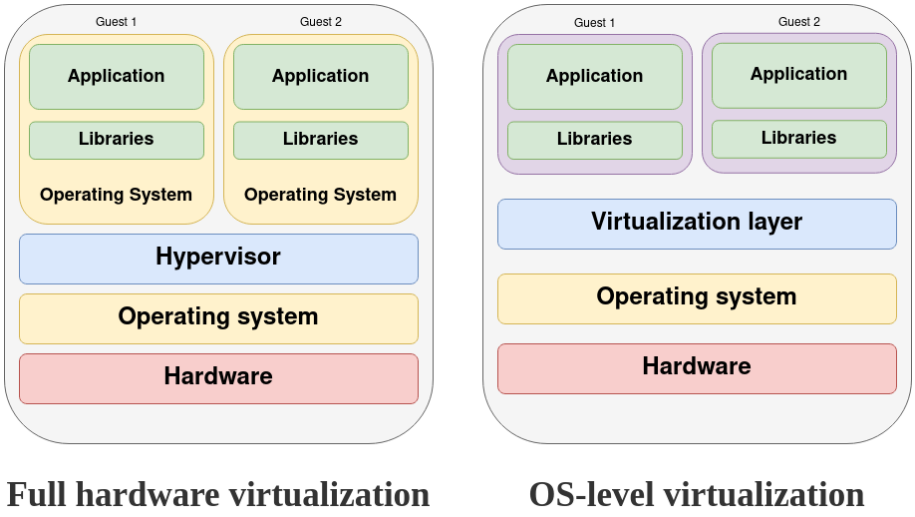

Figure 3.3: Full hardware vs OS-level virtualization

3.3 Docker

Docker itself consists of mostly three main parts. These parts are: *Docker engine*, *containerd* manager and *runc* runtime.

Docker engine primarily serves as application interface for user who wants to interact with *containerd*. *Containerd* is daemon responsible for running *runc* container runtimes. This hierarchy is illustrated in figure 3.4.

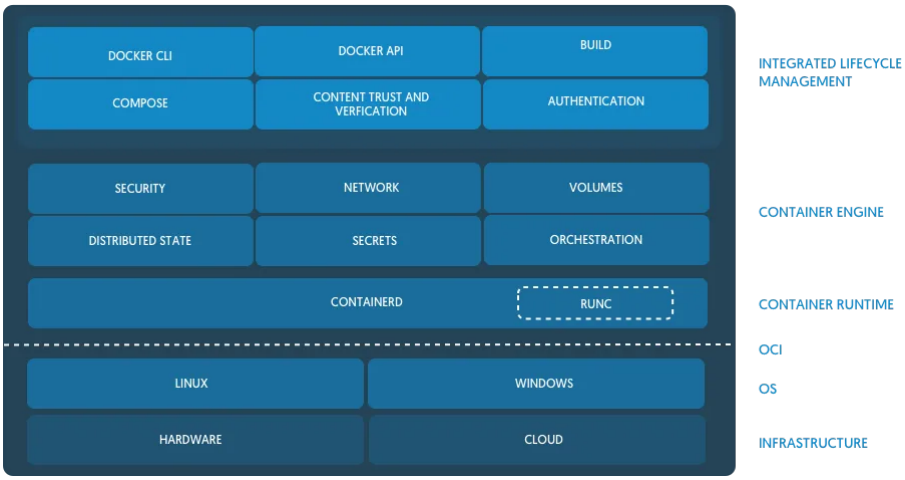

Figure 3.4: Docker engine - containerd - runc hierarchy. Taken from [1].

This runtime allows spinning up containers using *kernel namespace isolation* and *cgroups*. *Namespace isolation* separates groups of processes in a way that one group cannot see resources of another group. These resources can be processes, network interfaces, mount-points or inter-process communication[17]. Control groups (*cgroups*) is a kernel component that limits usage of resources (CPU, memory, disk I/O, networking etc.) of process groups [18].

There are more ways to communicate with Docker engine. Mainly it is docker *CLI* and docker *API*. API creates HTTP endpoints, via which any HTTP client can communicate with Docker engine. Docker CLI serves as an API client and prepares, then sends requests via socket, in Linux usually located at path */var/run/docker.sock*. These requests are same as requests sent via API. Docker CLI can also serve as an client for remote docker machine, which has Docker API enabled. This also means that any HTTP client capable of sending requests to Linux sockets is able to communicate with docker engine without docker CLI.

3.3.1 Docker concepts

Docker is based on Open container initiative (OCI) specification, which was established by Docker [19], therefore most concepts are interchangeable with OCI concepts, and usually docker is direct implementation of these concepts.

Docker image, or just image is a foundation of container. An image is an ordered collection of steps, or layers, and metadata which describes container runtime. These steps can be configuration changes, parameters for execution but mainly installation of application and its libraries. Images typically inherit layers of filesystems from other images. Dockerfile is used to describe image. Example of a simple Dockerfile, which packages Node.js app, is given in listing 3.1.

Listing 3.1: Example Dockerfile

```
FROM node:latest
COPY . /app
RUN make /app
ENTRYPOINT [ "node", "/app/app.js" ]
```

Layer is a collection of filesystem changes (addition, change or deletion of files) relative to its parent layer. Layers are used by layer-based or union filesystem to present one cohesive filesystem [20]. This can be achieved by filesystems such as AUFS or Btrfs.

As docker containers are a running instance of docker image and image itself is based on docker image, **size requirements** for containers might seem enormous. Especially, if there are more containers based on one image. This is solved by adding thin writable layer to every container, leaving other layers read-only. Writable layer is deleted when container is deleted. When layer

needs read access to a file, existing file is used. When a file residing in read layer needs to be modified (whether it is to change or delete it), copy is made to writable layer and this copy is modified [2]. Situation with more containers sharing one docker image is illustrated in figure 3.5.

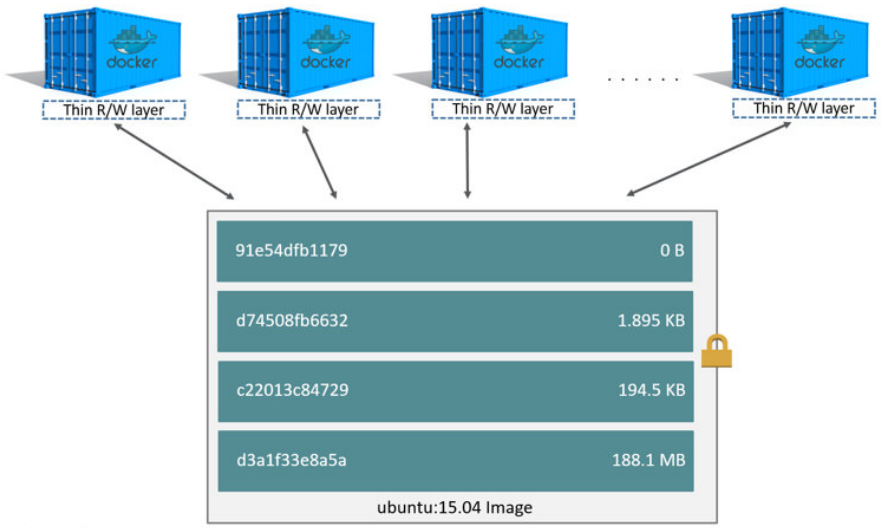

Figure 3.5: Docker containers sharing layers. Taken from [2].

By default, docker containers are started with restricted set of Linux kernel **capabilities**. This means that container cannot load or unload kernel modules, mount raw devices (including *cgroups* control devices) and others. Possible docker container capabilities are described in detail in [21].

Because data is written only in writable layer of a container, docker provides multiple ways to **store** this **data**. It can be a docker volume, a mount from host filesystem or a temporary filesystem. It may be used for persistence across container upgrades or to share data among containers themselves.

Docker registry is a hosted service which contains and provides docker images through Registry API. Default registry is available on Docker Hub (*hub.docker.com*) and can be browsed using web browser or using docker CLI [22]. Registry itself is an implementation of *OCI distribution specification* which specifies Registry API. There are many implementations of an API, which can be self-hosted, such as Distribution (Docker, [23]), Nexus Repository Manager 3 (Sonatype, [24]), Project Quay (RedHat, [25]).

3.4 Container orchestration

With rising number of containers, also rises the need to manage their lifecycles. This is especially true in a dynamic environment, where hundreds of containers are run. In this situation, container orchestration becomes essential.

Container orchestration is used to control and automate many tasks, mainly:

- provisioning and deployment of containers;
- redundancy and availability of containers;
- creating or removing containers to spread application load evenly;
- redeployment of container in case host becomes unavailable;
- resource management of containers;
- load balancing between containers;
- health monitoring;
- networking between containers [26].

There are plenty container orchestration engines, such as Docker Swarm, Kubernetes or Apache Mesos. This thesis describe Docker Swarm as it is used in software house environment specified in chapter 1.

3.5 Docker Swarm

Docker Swarm is a solution from Docker, used for orchestration of docker containers. Its architectural components consist of *manager nodes* and *worker nodes*, *stacks*, *services* and *tasks*.

Manager nodes are responsible for cluster management tasks such as maintaining cluster state, scheduling services, serving Swarm mode HTTP API endpoints. Manager nodes use Raft consensus algorithm to maintain consistent state of cluster, stacks and services.

Worker nodes have only one purpose, and that is to run containers. They do not participate in Raft consensus algorithm, make scheduling decisions or serve HTTP API endpoints [27]. Every manager node is by default also worker node. Therefore Swarm with only one node is possible. This can be useful for development purposes.

Atomic part of scheduling in Docker Swarm is a **task**[22]. Task can also be perceived as a container running on node specified by a service. Manager node assigns task to worker nodes according to a definition, which was provided in request to start a task. Tasks themselves cannot be defined as they are integral part of services.

Service is one or more instances (tasks) of an docker image. Service can run in two modes: replicated or global. In replicated mode, specified number of replicas are run in cluster whereas in global mode, container runs on every instance specified by restraints. Restraints can be labels or roles in docker Swarm (manager or worker). Additional resources can be assigned to services, such as secrets or networks. Services can be also grouped together in **stack**. To describe and distribute stack, **compose file** is used. Only services and stacks can be scheduled on *Docker Swarm*.

Chapter 4

Requirements for solution

Most of the requirements introduced in this chapter are based on software house environment described in chapter 1, but they are not limited to it.

The aim of this thesis is to find continuous delivery pipeline execution engine which fulfill requirements described in this chapter.

4.1 Containerization

Containerization is widely used technology across many different organizations. According to Portworx survey, 87% of respondents use container technologies [28]. As was stated in chapter 1, containers are widely used across production environment of a software house this solution is intended for.

For programs, containerization provide confined workspace with isolated process tree. This enables safer execution of arbitrary programs on infrastructure.

Programs which run in containers are well described by its container images. These images can be easily distributed via container registry and its deployment is specified by compose files or other declaration files.

Also lot of software already has containerized version, which can be run without complicated installation.

4.2 Serverless cloud computing execution model

There is no need to have always running instance of pipeline execution engine which waits for requests. This lack of necessity is justified by the fact that pipelines are not executed permanently and instance executing them would run all the time even when not utilized and resources allocated to it could have been used for other endeavors. This ineffectivity can be seen as both financial and environmental.

Serverless cloud computing execution model is a form of utility computing. Servers are still required for execution, but instance of application is not running all the time. When there is incoming request, application spins up, services incoming request and shuts down. If there are additional requests, it can spawn more instances of itself and serve rest of requests. [29].

Benefits of serverless execution model include: savings on costs as resources can be shared, delivery of updates is much faster, it is easily scalable and there is no need for dedicated maintainer of server which is running this solution.

4.3 Multitenant operational mode

Software house environment defined in 1 consists of multiple teams which might not event know each other, hence some form of multitenancy is required. According to Gartner, multitenancy refers to mode of operation of software where multiple independent instances operate in shared environment[30]. In this particular case, instance can be seen as workspace of a team or a project and environment is infrastructure of framework for CI/CD.

Solution must enable access control where entity is project and identities accessing it are developers. Projects also cannot access other projects resources such as workspaces or credentials to access other resources. Overall *isolation* of projects and identities is required.

Common practice in multitenant environments is to limit access to resources. This means that pipeline cannot allocate more resources than there are declared in pipeline specification.

Also some form of accountability is required for usage statistics. Another characteristic apart from isolation is self serviceability described in following section.

4.4 Self-serviceability within project teams

Solution is intended for small independent teams in larger-scale organization. This means that these teams should be able to create and maintain pipelines for projects on its own and not depend on one team which will support all pipelines.

This self-serviceability should be intuitive and easy to use, otherwise it can be prone to errors or other ineffectiveness.

4.5 Open Source

Using open source projects as a foundation for solution provides many benefits. Open source software provides control over itself with ability to examine source code and ensure what does software do. This also means that bugs are easier to detect and to pinpoint its origins and fix them. Economic aspect of

open source is considerable as usage of it enables to shift costs from licensing to customization and implementation itself. Also customization itself is big benefit for instances when some functionality is not exactly as required, it can be modified.

Open source licenses in general are implied with no particular license requirements. Licenses listed by Open Source Initiative [31] such as Apache License 2.0 or MIT license are preferred in solution which will be proposed in this thesis.

4.6 Integration with the existing services

As numerous services exist in environment specified in chapter 1, solution is ought to be able to integrate with them.

Starting point of a delivery pipeline is a source code, hence integration with source code repository manager is required. Git technology is primary source code management solution in software house environment, therefore solution should be able to access external git repository specified by its link and credentials used to access it.

Projects have dependencies which might not be accessible without authentication, and also artifacts created in integration and delivery process are meant to be stored in repositories (docker images, CDN libraries, or any arbitrary output), thus solution must have a way to provide arbitrary credentials to it.

As git can have hooks installed, pipeline must be able to be started from URL endpoint and also upon its completion some form of callback should be possible.

In software house environment, central identity provider is used. This provider is based on Open ID Connect technology, hence access control is provided by identities from this provider.

4.7 Simple yet powerful

As was already explained in chapter 1, significant proportion of software coming from this particular environment is based on standardized web application framework, therefore standard usage should be as easy as specifying git repository and branch to build. Everything is then done automatically without the need of a user input.

On the other hand, considerable amount of projects require non-standard software. For example, it can be C library or Go integration application. This minority of projects however should not have special treatment as this brings unnecessary complexity to maintenance, as well as it can introduce security risks. Therefore possibility to accommodate non-standard uses is not difficult.

4.8 Design of metrics

Metrics used for technology research, are described in this section. These metrics will serve as foundations in Research of technologies described in chapter 5.

4.8.1 Declaration of a pipeline

Pipeline should be defined as a code, which means it is tracked in some form of revision system and it enables to track which pipeline was executed.

The way in which pipeline is defined is important. This means that requirements for creation of a new pipeline must be listed. This includes format it uses.

User knowledge of containers must be defined to correctly estimate complexity of creating new pipeline. This knowledge can be divided into four levels:

- No experience - user does not need any experience with running containers.
- Basic - user knows what are containers.
- Medium - user knows how to build containers.
- Advanced - user has knowledge of container networking, persistence, etc.

Resources, which must be defined by user in order to use them in pipeline need to be described. This is mainly oriented towards credentials used to access external resources. These resources, along with pipeline itself, should be scoped on specified entity, such as group, namespace, or other to provide some form of multitenancy.

All of mentioned above can be summarized into following criteria:

- Support for ”pipeline as a code” definition.
- Required user experience.
- Scripting experience.
- Configuration format used to define pipeline.
- Self serviceability - how much can user define without need of maintainer of solution to intervene.

4.8.2 Usage of containers

How are containers used in solution must be defined. Whether containers are used only as means of distribution of a solution or build itself is executed in containers. Also if a method, in which containers are used, provides secure way to define its execution. This means that how complex, or if even possible, it is to gain access to underlying infrastructure if user with malicious intents has access rights to execute pipeline.

These criteria are analogous to user knowledge defined in previous section:

- Basic - user can execute containers with some parameters.
- Medium - user gains access to execution environment of a container.
- Advanced - user can set up container networking, persistence, enable or disable container capabilities, etc.

4.8.3 Access control

Access control should be present in solution as it will be used by numerous users across whole software house environment, which is described in chapter 1.

To evaluate this, following criteria are used:

- Attribute-based access control.
- Role-based access control.
- No access control.

4.8.4 Integration

Identities and its access rights must be sourced from external services which support *OAuth2.0* authorization.

As was mentioned in section 4.6, integration with existing git repository is required.

Also integration with Docker Swarm orchestration is required as it is orchestration engine in software house environment(1). This integration must be native without workarounds.

4.8.5 Resource requirements

One of the metrics are resources required to run the solution. These include processing power, memory and storage.

4.8.6 Scalability

As this solution will be used by high number of developers, scaling is also one of the requirements.

To understand possibilities of scalability, how are pipelines executed on infrastructure layer must be described.

Solution should be scalable in both vertical and horizontal manner. Vertical scaling means that more resources application has available, more of them it can use. Horizontal scaling means that more instances of the same application are added thus increasing performance of application.

Following criteria can be derived from these two methods of scaling: *vertical*, where it is not possible to add more instances to existing one; *static horizontal*, where instance must be reconfigured to be scaled accordingly; and *dynamic horizontal*, where no instance of a solution is running, when no execution is requested and if such request appears, it can be scaled accordingly.

4.8.7 Extensibility

Possibility of extending existing solution must be described. This include description of what can be extended. *Execution inside pipeline*, for example different behavior if pipeline fails; *execution outside pipeline*, for example authentication, authorization or audit; or *tooling* used during pipeline execution, such as version of compiler or test suite used.

And also complexity of such extension should be described. Whether solution does have defined API for extending or if extensions can be plugged in or out.

4.8.8 Summary of metrics

Sections, which were mentioned above, describe metrics which will be used to evaluate technologies, to help with this evaluation, table 4.1 was created.

Table 4.1: Metrics used to evaluate researched technologies

Metric	**Criteria**		**Evaluation**
Declaration of pipeline	Pipeline as a code		Yes/No
	User knowledge		None,Basic,Medium,Advanced
	Declaration format		JSON, YAML, arbitrary
	Self serviceability		Yes/No
Usage of containers	Basic		Yes/No
	Medium		Yes/No
	Advanced		Yes/No
Access control	Attribute-based access control		Yes/No
	Role-based access control		Yes/No
Integration	Authentication		Yes/No
	Authorization		Yes/No
	External git repository		Yes/No
	Docker Swarm		Yes/No
Scalability	Static horizontal		Yes/No
	Dynamic horizontal		Yes/No
Extensibility	Options	Inside pipeline	Yes/No
		Outside pipeline	Yes/No
		Tooling	Yes/No
	Means	API	Yes/No
		Plugins	Yes/No

Chapter 5

Research of technologies

This chapter is dedicated to the research of technologies available, which could match requirements and metrics defined in chapter 4.

Jenkins, *GitLab* and *Tekton* were chosen. Choice of these technologies was based on landscape provided by Continuous Delivery Foundation [32].

Jenkins is widely used solution across software house environment defined in chapter 1. Many developers are familiar with its concepts, therefore, it is reasonable to investigate possibilities it is offering.

Next in list of technologies which will be researched is *GitLab*. It is also widely known for its Continuous integration and delivery capabilites.

Last but not least is *Tekton*. Project developed mainly by Google as an open source tool for running pipelines in container orchestration engine Kubernetes.

5.1 Jenkins

Jenkins is a self-contained, open source automation server which can automate all sorts of tasks related to CI/CD [33] licensed under MIT License [34].

It is widely known service with more than 270 000 running installations and more than a 1 000 000 execution nodes as of March 2020 [35].

Architecturally, Jenkins is made up of two main components, *master* and *agent* servers. **Master** server is responsible for managing and controlling execution of *build jobs*. It contains web portal, from which Jenkins is managed and operated. Most operational tasks can be also performed via REST API. For more advanced configuration, Groovy CLI can be used. **Agents** are nodes on which execution of jobs takes place. They can have specific labels to be distinguishable from other agents, for example `Windows` or `GPU`.

Build job is an essential part of Jenkins build process which can be thought of as a particular task or step, or whole project build in a build process. It can range from *shell* script, *maven build* project, *pipeline* build and many more with a help of plugins.

It is written in Java with Groovy as a main scripting language, which allows to run arbitrary Groovy scripts within the Jenkins master runtime or in the runtime of an agent [36, chapter Managing Jenkins].

There is also *JenkinsX*, which is derived from *Jenkins*, but it only supports Kubernetes orchestration engine and uses Tekton to do so.

Summary of metrics discussed in subsequent sections is available as table 5.1.

Table 5.1: Summary of metrics for *Jenkins*

Metric	**Criteria**		**Jenkins evaluation**
Declaration of pipeline	Pipeline as a code		Yes
	User knowledge		None, Basic, Medium, Advanced
	Declaration format		Jenkinsfile, Groovy script
	Self serviceability		No
Usage of containers	Basic		Yes
	Medium		Yes
	Advanced		Yes
Access control	Attribute-based access control		No
	Role-based access control		Yes (by plugins)
Integration	Authentication		Yes (by plugins)
	Authorization		Yes (by plugins)
	External git repository		Yes
	Docker Swarm		Yes (by plugins)
Scalability	Static horizontal		Yes (agents only)
	Dynamic horizontal		No
Extensibility	Options	Inside pipeline	Yes
		Outside pipeline	Yes
		Tooling	Yes
	Means	API	Yes
		Plugins	Yes

5.1.1 Plugins

First of all, Jenkins plugins must be described as most of the functionality of Jenkins is provided by them. This ranges from defining credentials to access control.

Plugins are primary means of extending functionality of Jenkins to suit user-specific needs[36, chapter Managing Jenkins]. There are more than thousand available open source plugins [37].

They can vary from wrapping API of different service to be usable in build jobs, enriching Jenkins with statistics and usage graphs, ability to configure Jenkins via declarative configuration files to extending functionality of Jenkins with Jenkins Pipelines.

Plugins are distributed via Update center, which is hosted inventory of plugins and its metadata with dependencies, versions etc.

To install plugins, one can use Update center or install plugins manually on master via `.hpi` plugin files.

5.1.2 Declaration of pipeline

Pipeline in Jenkins is declared as file which usually resides in git repository.

Step is a fundamental building block of a pipeline. It tells Jenkins what to do, e.g. retrieve source code from VCS, run maven build, upload built artifact to repository, etc. and where to do it, for example, only agent nodes marked with `GPU` label.

Steps can be grouped together in **stages**, which serve as an abstract separation for tasks which belong together. Stages can also contain metadata for steps such as: agent node, on which execution will take place; environment variables available for steps, and other. Examples of a stage are: build application, run system tests, promote built artifact to release, etc. Files describing Jenkins pipelines are called Jenkinsfiles.

To create a new pipeline, user should be familiar with Groovy scripting language or use Jenkins's own domain specific language (DSL), which has structure similar to JSON.

Declaration of a pipeline usually resides in git repository. This repository has to be accessible by Jenkins (that means private key to access non-public repositories is required).

To access secured resources, *credentials* must be created in *global configuration* of *Jenkins*. All other credentials to access resources outside of Jenkins such as Software repository, Docker registry, etc. are created this way. This means that users without administrator access rights are unable to create such credentials.

5.1.3 Usage of containers

Containers in Jenkins can be used as an execution environment instead of standard *agent* executors. This means that steps that were previously executed as standard shell scripts at execution environment of an agent are now executed inside a container.

It does not limit user in supplying specific arguments to docker daemon, such as executing container with privileged flag, which disables all security measures prepared by docker engine or mounting whole host filesystem.

5.1.4 Access control

Access control in Jenkins can be controlled in two axes [36, System administration, Securing Jenkins] listed below.

- Security realm, which determines users and their groups.
- Authorization strategy, which determines who has access to what.

Both of these can be modified by numerous plugins, such as *GitHub Authentication* [38] or *OpenId Connect Authentication* [39] for security realm, or *Authorize Project* [40], *Role-based Authorization Strategy* [41] or *Matrix Authorization strategy* [42].

5.1.5 Integration

As was mentioned in previous section, integration with Identity provider can be achieved by multiple plugins, such as *OpenId Connect Authentication*. To authorize user, *Matrix Authorization strategy* can be used. This enables fine grained permissions for projects such as viewing and starting pipeline.

Git repository integration is provided by creating credentials which have access to external git repository.

Swarm integration can be achieved by Docker Swarm [43] plugin. This plugin enables to spinning up *Jenkins agents* in Docker Swarm. Every agent image must be manually preconfigured in *global plugin configuration*. This means that if new image is required to start, service intervention is required. Configuration requires *Docker Swarm manager* address and TLS certificate for authentication to manager.

5.1.6 Resource requirements

Jenkins memory requirements can range from 200MB to 70+GB of RAM for a single Jenkins master server. Also, each connection to agent node can take at least 2MB of memory [36, Hardware recommendations].

Recommended hardware configuration for small team consists of 1 or more GB of RAM and 50 or more GB of drive space [36, Installing Jenkins]. No specific requirements for processing power are set but it must be noted that Jenkins master will serve all HTTP requests to access Jenkins and also orchestrate execution of pipelines on agents.

5.1.7 Scalability

As was stated in introduction in 5.1, execution model consists of *Master* node, which schedules pipelines on *Agent* node. If connection to agent is lost, *master* schedules build of a pipeline on another available agent. Downtime of master node means that no pipelines can be run.

Scalability in Jenkins is mainly achieved by connecting more agents to master server. Recommended number of jobs per master is one hundred jobs per core. Unfortunately, Jenkins does not support horizontal scaling of masters and when scaling is required, new instance of Jenkins is required [36, Architecting for Scale].

5.1.8 Extensibility

As was described in section 5.1.1, Jenkins is highly extensible by plugins. Jenkins pipelines are also extensible by using shared libraries, which extend functionality of pipelines itself. Plugins must be written in Java language but shared libraries can be also written Groovy.

5.2 GitLab

GitLab is a web-based development and operation tool that provides git repository, container registry and also offers continuous integration and delivery tooling. This is known as *GitLab CI/CD*.

GitLab comes in two editions, Community and Enterprise. In this section Community edition will be taken into account as it uses MIT license which conforms requirements on licensing described in section 4.5.

Pipeline in *GitLab CI/CD* can be divided into multiple jobs. Job is fundamental element of GitLab pipeline declaration. It is top-level element which contains at least *script* clause. This clause defines what shell commands are executed in a job context [44, CI/CD, .gitlab-ci.yml Reference].

Summary of metrics discussed in subsequent sections is available as table 5.2.

5.2.1 Declaration of pipeline

Pipeline in GitLab is declared in YAML configuration file, `.gitlab-ci.yml` which resides in git repository of a project, which pipeline builds.

Declaring pipeline is straightforward. Pipeline YAML configuration file is created. In this file, stages are declared from jobs. Jobs declaration is also present in same configuration file.

Git repository, in which pipeline declaration is, must be present on GitLab server. This means that no additional credentials to access git repository are required.

Other secured resources, such as credentials, must be defined as variables in project settings. This means that credentials used to access secured resources outside GitLab are available as an environmental variable to all processes running in job.

These resources are bound to project, on which access rights can be managed. Access rights can be also managed on groups. Therefore multitenancy is possible.

5.2.2 Usage of containers

Usage of containers is supported in GitLab pipelines. Image can be specified per whole pipeline as well as per job in pipeline. Specifying any arguments to

Table 5.2: Summary of metrics for *GitLab CI/CD*

Metric	Criteria		GitLab CI/CD evaluation
Declaration of pipeline	Pipeline as a code		Yes
	User knowledge		None
	Declaration format		YAML
	Self serviceability		Yes
Usage of containers	Basic		Yes
	Medium		Yes
	Advanced		No
Access control	Attribute-based access control		No
	Role-based access control		Yes (Only predefined groups)
Integration	Authentication		Yes
	Authorization		No
	External git repository		No
	Docker Swarm		No
Scalability	Static horizontal		Yes
	Dynamic horizontal		No
Extensibility	Options	Inside pipeline	No
		Outside pipeline	No
		Tooling	Yes
	Means	API	No
		Plugins	No (only *File hooks*)

docker itself is not enabled. This means that volumes or flags to containers itself such as privileged flag are impossible. Scripts are then executed in container.

5.2.3 Access control

Permissions in *GitLab CI/CD* rely on roles user has in GitLab, four roles are present in total; admin, maintainer, developer, guest/reporter [44, User, Account, Permissions]. Admin role has access to all pipelines in scope of whole GitLab instance. Maintainer can create, remove and change pipelines and jobs, developer is able to only execute pipelines and remove job artifacts and guest/reporter can only see commits and jobs.

5.2.4 Integration

GitLab can be configured to use OpenID Connect provider to retrieve identities. Assigning groups from identity provider token is not supported and users must be invited to groups or request access to it. Therefore, external authorization using OpenID Connect Identity provider is not supported.

Community edition of *GitLab CI/CD* does not support creating pipelines from external repositories, such as GitHub, Bitbucket or arbitrary git repository [44, CI/CD, External repositories].

Swarm integration is also unavailable [45].

5.2.5 Resource requirements

To run GitLab server, at least 2 cores and 8GB of RAM are required [44, Install, Requirements]. These requirements only cover master server which serves mainly for scheduling jobs and serving front end of *GitLab CI/CD*. These requirements are for instances for up to 100 users [44, Administrator, Reference Architectures]

There are no specific requirements for *GitLab Runner*, which executes jobs from pipelines.

5.2.6 Scalability

GitLab server is required to run at all times. Execution of pipelines in GitLab is provided by *GitLab Runner*, which is compiled binary that executes jobs in pipelines. This binary registers to GitLab server and retrieves jobs definitions which are then executed. This can be SSH command, shell, Docker, Kubernetes and many others [46, Executors].

GitLab runners can be added during runtime of GitLab server. This means that horizontal scalability of execution agents is provided by adding more GitLab Runners.

GitLab master server can run on one host but its deployment can be switched to distributed, where multiple services which form up GitLab run. In this type of deployment, GitLab supports up to 50 000 users[44, Administrator, Reference Architectures]. This scaling is static and does not support serverless execution model.

5.2.7 Extensibility

As containers can be used to execute specific *jobs* in *GitLab CI/CD* pipeline, tools used inside pipelines can be extended (or switched) by using different images of containers.

GitLab offers possibility to execute action on specific event called *File hooks*, these events can be for example actions around project or user such as creation, deletion, rename etc. Various other system hooks exist, they are available at GitLab Docs [44, Administration, Configure, System hooks]. File hook can be any executable file which consumes JSON from standard input.

Other than options mentioned above, there is no other way of extending GitLab present except for editing source code itself.

5.3 Tekton

Tekton is a framework for creating continuous integration and delivery pipelines. It is licensed under Apache License 2.0. *Pipelines* in Tekton consist of multiple tasks. *Tasks* are main building blocks and are comprised of steps as containers. Parameters are passed to steps but no additional execution is taken inside containers in contrast to *GitLab CI/CD, 5.2.2* or *Jenkins, 5.1.3*.

Pipelines and *tasks* are defined in advance, and runs itself are called *PipelineRun* and *TaskRun* respectively.

It is deeply integrated with Kubernetes orchestration engine, therefore, a lot of its aspects are delegated to Kubernetes. Tekton itself is possible thanks to leveraging custom resource definitions which are provided by Kubernetes to extend Kubernetes API[47].

Summary of metrics discussed in subsequent sections is available as table 5.3.

Table 5.3: Summary of metrics for *Tekton*

Metric	Criteria		Tekton evaluation
Declaration of pipeline	Pipeline as a code		Yes
	User knowledge		Basic, Medium, Advanced
	Declaration format		YAML
	Self serviceability		Yes
Usage of containers	Basic		Yes
	Medium		No
	Advanced		Yes
Access control	Attribute-based access control		Yes
	Role-based access control		Yes
Integration	Authentication		Yes
	Authorization		Yes
	External git repository		Yes
	Docker Swarm		No
Scalability	Static horizontal		Yes
	Dynamic horizontal		Yes
Extensibility	Options	Inside pipeline	No
		Outside pipeline	No
		Tooling	Yes
	Means	API	No
		Plugins	No

5.3.1 Declaration of pipeline

Pipeline is defined as an ordered set of *Tasks*. Its definition is saved as an YAML file. Tasks definitions are located in additional YAML file. These files

are then sent to Kubernetes via standard ways of creating resources in cluster.

User must be be acquainted to way Kubernetes resources are declared.

To use secured resources, Tekton offers two builtin types of authentication. First is basic authentication which consists of username and password, and the other one is SSH private key authentication. To use other types of credentials, they first must be declared in Kubernetes and then mounted as a volume to a container in *Task* declaration.

These resources are bound to namespaces in Kubernetes, on which access rights can be managed. Therefore some form of multitenancy is possible.

5.3.2 Usage of containers

Tekton natively uses containers as its pipelines are executed in container orchestration engine. Raw arguments to container execution cannot be passed, but security context can be changed to privileged, and also bind mounts are possible.

5.3.3 Access control

Authentication and authorization is provided by Kubernetes. As in this thesis' scope, *Docker Swarm* is an orchestration engine, it will not be described in great detail. Briefly, user must be authenticated to call Kubernetes API requests. There are multiple authentication methods, more about them in [48, Accessing the API, Authenticating]. Authorization can have multiple modes (RBAC, ABAC) which is tied to namespace in which pipelines are run. Details can be studied at [48, Accessing the API, Authorization Overview].

5.3.4 Integration

To issue requests to run Tekton pipelines, Kubernetes authorization is required. Integration is possible by using *client-go* credential plugins. It can be configured in such a way, that it prompts user to log in at identity provider. From provider, it retrieves token and sends this token to Kubernetes which will then submit token to identity provider. Identity provider then verifies this token and returns user's username and groups.

To execute Tekton pipelines with existing git repository, credentials to remote repository are required. Otherwise, no other integration steps are required.

Swarm is not supported as an orchestration engine, as Tekton's deep integration with Kubernetes orchestrator.

5.3.5 Resource requirements

Because Tekton itself does not serve requests by itself, it does not have any specific resource requirements. On the other hand, as Tekton is highly cou-

pled with Kubernetes cluster, resource requirements for Kubernetes are worth noticing.

Kubernetes cluster requires at least 2GB of RAM per machine and 2 CPUs [49] only for its runtime, therefore more resources are required for an actual execution of pipelines.

5.3.6 Scalability

Tekton itself is not permanently running and uses serverless execution model. This means that it natively support scaling. Therefore only limitation is scalability of Kubernetes cluster underneath.

5.3.7 Extensibility

Tekton steps consist of containers which are then executed in *TaskRuns*, therefore tooling modularity is provided by default.

Tekton currently does not support Pipeline extensibility [50], nor Task extensibility [51].

Chapter 6

Evaluation of technologies

Jenkins is widespread standard [35], which laid foundations for pipeline as a code [52]. It offers great deal of extensibility with plugins and its idea of using shared libraries satisfy DRY (Do not Repeat Yourself) principles. As process of building and releasing software in software house environment 1, is mostly the same and projects have its predefined structure, shared libraries can serve as a basis for custom framework which would implement individual steps from pipeline such as build, test, release, etc. Unfortunately, Jenkins architecture is not dynamically scalable and does not support serverless execution model. Also access control is only supplied by plugins not built into core. Therefore, to maintain this solution and operate it, is deemed inefficient.

GitLab is complex solution providing not only pipeline orchestration engine but also git repositories, request management, docker registry and other. But these features are unusable in software house environment 1, as other features it provides are already present in this environment.. Also inability to use external repositories proved *GitLab* as inapplicable for proposed solution.

Tekton is the youngest of the compared technologies and with its architecture, it best supports containerization and scalability. Its scalability supports serverless execution model and modular execution of containers allows to start images, which might not have user shell available in containers environment. Due to Tekton's deep integration and reliance on Kubernetes, it does not allow deployment to *Docker Swarm*.

Among all the technologies, Tekton best fits the requirements given in chapter 4. Unfortunately, it is not currently possible to switch software house environment container orchestration engine from *Docker Swarm* to *Kubernetes*.

Jenkins is considered as second best fit. It is already used in software house environment, but as was stated before, its architecture does not enable scalability and management of access control is unsustainable with growing number of users.

Since Docker Swarm is more of a technology in decline [53], and Kubernetes

on the rise [54], it can be expected that software house environment will eventually move to Kubernetes. When this happens, Tekton will be the most fitting technology.

Because aim of the thesis is to provide prototype of continuous integration and delivery pipeline, option of combining principles mentioned above is investigated. Core idea is to use Jenkins' extensibility with Tekton's serverless execution model which utilizes underlying container orchestration engine. This is possible by using *Jenkinsfile runner*, which will be described further in section 7.1 in conjunction with *Jenkins shared libraries* and capabilities of *Docker Swarm*.

By using these principles, eventual switch from *Docker Swarm* to *Kubernetes* and Tekton can be relatively simple.

Chapter 7

Proposed prototype design

General idea of the design proposed in this chapter, is that *Jenkins* will not be installed on server but will still serve as an pipeline execution engine. It will execute pipeline, which is defined as a file in git repository. This means that every time, request to execute arrives, container of *Jenkins* is started, which then begins to perform tasks and steps defined in pipeline. Individual tasks defined in pipeline are then executed as separate containers in Swarm cluster. This allows flexibility as provided by *Tekton* with only difference that instead of *Kubernetes* orchestration engine, *Docker Swarm* is used.

For a detailed description of the prototype, it is necessary to explain several principles such as shared storage over which individual steps of pipeline operate and credentials used to access external secured resources, like git repositories.

7.1 Principles

7.1.1 Docker storage

By default all data written by docker containers is stored in writable layer of a container. As was mentioned in section 3.3.1, if container is deleted, writable layer is also deleted. This means that all data created by container is lost. Also data cannot be shared between containers.

Docker has three options of persisting and sharing data across containers: *volumes*, *bind* mounts and *tmpfs* mount. [55]

Volumes are a type of storage, where files are stored in Docker engine managed area, which can be also described as installation directory. This directory is, on Linux, usually located in path */var/lib/docker*. These files should not be modified by processes that are not related to docker. According to Docker, it is preferred way of sharing files and persisting data across container restarts.

Bind mounts are files or directories, mounted directly from host filesystem to container. It can be mounted from anywhere on the host. This can be useful

for sharing configuration for programs such as *.ssh* directory or */etc/hosts* file.

Temporary file system, or *tmpfs* for short, is functionality provided by Linux kernel. It enables to store files in memory. This can be useful for data which is not desired to be persisted either on host or within container, such as credentials and such.

7.1.2 Docker secrets

Docker secret is a functionality of a *Docker Swarm* orchestration technology. It enables to share data which is not meant to be transferred over network or stored unencrypted in Dockerfile or source code repository. This data can be passwords, SSH private keys, SSL certificates or any other file or data which cannot be passed to container via environment variable.

This data is then mounted as *tmpfs* to target container. Secrets cannot be used in standalone containers, but only with *Docker Swarm*.

7.1.3 Jenkins Pipeline

Jenkins pipelines can be written in *scripted* or *declarative* style, where both types are based on Groovy engine as a foundation. Specific steps can be used in scripted and declarative pipelines which share "pipeline as a code" principle, where pipeline is defined in VCS and every change is versioned. Main difference is in syntax where scripted pipeline follows imperative programming model and declarative pipeline declarative programming model.

Scripted pipeline is much more reminiscent of pure Groovy syntax with variable definitions, control flow elements such as loops and conditions, objects declaration and many more. This provides great amount of flexibility and extensibility, but as Groovy learning curve is not always desired, declarative pipeline was created [36, chapter Pipeline]. Example of Jenkins scripted pipeline can be seen in listing 7.1.

Listing 7.1: Jenkins Scripted Pipeline

```
node {
   def maven_version = "3" + "-alpine"
   node("maven:${maven_version}") {
      stage('Example Build') {
         echo 'Hello Maven'
         sh 'mvn --version'
      }
   }
   for (def i = 0; i < 10; i++) {
      def version = ""
      if ( i % 2 ) version = "8-jre"
      else version = "8-alpine"
      node("openjdk:${version}") {
         stage("Example Test ${i}") {
            echo 'Hello, JDK'
            sh 'java -version'
         }
      }
   }
}
```

Declarative pipeline has simpler and easier to understand, but much stricter syntax and pre-defined structure. It is much better suited for simpler continuous delivery pipelines. Example of Jenkins declarative pipeline can be seen in listing 7.2.

Listing 7.2: Jenkins Declarative Pipeline

```
pipeline {
   agent none
   stages {
      stage('Example Build') {
      agent { docker 'maven:3-alpine' }
      steps {
         echo 'Hello, Maven'
         sh 'mvn --version'
      }
   }
      stage('Example Test') {
         agent { docker 'openjdk:8-jre' }
         steps {
            echo 'Hello, JDK'
            sh 'java -version'
         }
      }
   }
}
```

7.1.4 Jenkins shared libraries

Another means of extending functionality of Jenkins pipelines, otherwise than plugins, is by creating shared libraries. This can be especially useful when patterns among pipelines emerge, as they are used for sharing its reusable parts, or functionality, across projects in an organization.

As Jenkins pipelines are based on Groovy foundations, shared libraries are also written in Groovy language. In fact, declarative pipeline model definition is mostly written in Groovy in a manner similar to shared libraries [56].

Shared library is defined by its identifier, its link to VCS, default version and optionally VCS credentials to access it. Identifier is then used in conjunction with version in pipelines. Version can be anything what VCS can understand, such as git commit hashes, tags or branches.

Specific directory structure for shared library is required by Jenkins, to understand library and correctly load it. It can be described as follows:

```
root
├── src ..................................... Groovy source files
│   └── org
│       └── foo
│           └── Bar.groovy ................... for org.foo.Bar class
├── vars
│   ├── foo.groovy . global variable 'foo', available as a step in
│   │   pipeline
│   └── foo.txt ..... help for 'foo' variable
└── resources ..... resource files
    └── org
        └── foo
            └── bar.json ..... static helper data for org.foo.Bar
```

Variables become available as pipeline steps under file name, specified in `vars` directory. Only files with `.groovy` extension following `camelCase` convention are loaded.

Functions defined in `.groovy` files are available in pipelines by dot-notation. For example, if in file `vars/log.groovy` is a function `def info(message)`, it is accessed in pipeline as `log.info "Hello World"`. Default function which is executed on call is named `call(...)`.

Libraries can be loaded explicitly at a start of a pipeline, statically by `@Library('id@version')` notation or dynamically by `library(...)` step, anywhere, where steps are valid.

Another way to load libraries is to load them implicitly. This can be useful if it is desired to alter functionality of existing steps, or to pre-define version of library, which will be used. Implicit loading of libraries is set up in Jenkins configuration of a shared library, where identifier and VCS link is defined.

More about Jenkins shared libraries can be read at [36, chapter Pipeline, section Extending with Shared Libraries].

7.1.5 Jenkinsfile runner

Jenkinsfile runner is an experimental command line tool which packages Jenkins to provide pipeline execution environment without the need of a master server.

This means it can be used in executing Jenkins pipelines in a confined ephemeral environment, such as containers, without need of exposing ports or complex management of an always running master node. It can also be used for assistance in editing Jenkinsfiles locally and run integration tests of shared libraries.

Jenkinsfile runner utilizes only `jenkins.war` file, which is core of a Jenkins, thus no web user interface is available. Therefore, plugin installation and configuration is achieved by other means.

Plugin installation can be accomplished by manual installation of plugin `.hpi` files, but this can be tedious as many plugins require additional plugins as a dependency. To mitigate this problem many helper tools were created such as `install-plugin.sh`, available in docker distribution of Jenkins [57] or *Plugin Installation Manager Tool*, which goal is to replace other plugin installation tools [58].

Configuration is achieved with help of plugins, specifically *Configuration as Code Plugin*, which provides a way to serve a human-readable declarative configuration for Jenkins. This is mainly used to declare shared libraries, configuration for plugins and credentials available in pipelines.

With use of *plugin installation tool* and *Configuration as a Code plugin* it is possible to prepare this application in a container image for easier distribution and faster setup.

More documentation and information about Jenkinsfile runner can be found at its source code repository [59].

7.1.6 Tekton tasks

As was mentioned in section 5.3, Tekton pipeline consists of *Tasks*. Task then executes containers as individual steps.

Task runs on a Kubernetes *pod*. In short, *pod* is a group of containers running together on one host as if they were on same machine. This means that they share Linux namespaces and all have access to shared volumes and services within pod are available at *localhost* hostname.

This enables to share workspace of steps which are tightly coupled together, such as unit tests, compilation and packaging. On the other hand, separate tasks do not need to run on same underlying host and can be run across whole cluster to evenly distribute load of a pipeline.

7.2 Architecture

This section contains description of components used in architecture of a proposed prototype, together with their roles. Next is described topology of components and their interaction between each other in 7.2.2. Last but not least, principles used in design are described in section 7.2.3.

7.2.1 Components

Core components and their role in overall architecture are described in this section.

7.2.1.1 Task

Task is main building block of a pipeline. It is analogous to *stage* in Jenkins pipeline, *job* in *GitLab CI/CD* or *Task* Tekton. Task defines multiple *steps* which are fundamental in pipelines. Steps perform single action such as checkout from source code, execution of a script etc.

Main difference between *task* and *stage* from Jenkins pipeline is that steps in *tasks* are executed as containers on one host in *Docker Swarm*. It provides separation of pipeline from an underlying infrastructure. Running on one host is also advantageous as it provides shared workspace between individual steps because local *docker volumes* cannot be shared across cluster and are bound to one host.

7.2.1.2 Task executor

Task executor is an application running on a *Swarm node*. It is responsible for execution of tasks. Input is a list of steps to run with additional information such as volumes to mount, environment variables, and so on.

Steps cannot be direct *docker* binary commands to create containers, as it would enable bind mount filesystem of an underlying host or add Linux capabilities such as load and unload kernel modules, access device and many more.

Therefore, only arguments to containers itself can be arbitrary and everything else, such as volumes, environment variables etc. must be included in additional structured information.

To execute steps, task executor requires docker daemon, by which containers are spawned. Therefore docker socket is mounted to task executor.

This is different than creating Docker-in-docker, where container run by docker daemon spawns another docker daemon to execute containers in. To enable this, container, which will run daemon must be started with privileged flag, which effectively disables security measures [21].

Hence Docker-on-docker principle is used. This forwards *docker socket*, used for control of a *docker daemon*, to container, which can then issue re-

quest to Docker API on host it is running. As this container's only input is task specification with predetermined structure passed to container as a environmental variable, attack surface on this exposure is narrow.

7.2.1.3 API server

API server handles compose files which define stacks running on *Docker swarm*. These stacks can be *tasks*, *pipelines* itself or any arbitrary compose file provided.

Compose files were chosen for declaration of services as they are used to declare *Docker Stacks*. They enable deployment of multiple services with additional resources defined, such as volumes, secrets and many more. Docker compose file format is not bound to *Docker Swarm* and can be used in local developer environment. Additionally, compose files can be easily parsed and converted to describe services in other container orchestrator, such as Kubernetes [60].

Handling of compose files is required as *Docker Swarm* does not support deploying stacks via API. Parsing compose files is also required as it might contain bind to docker socket, or enable containers various kernel capabilities such as mounting devices.

Also authorization whether user sending requests can execute pipeline in specified namespace is done at *API server*. Claims to resources are provided by *Reverse proxy*, described in 7.2.1.4.

7.2.1.4 Reverse proxy

Reverse proxy operates as a server which inspects requests incoming to API server and authenticates them against *OAuth 2.0* or *Open ID Connect* identity provider.

Claims supplied in token provided by identity provider are forwarded to *API server* as means to authorize incoming requests. These claims must contain authorization privileges for user, who sent the request.

7.2.1.5 Jenkinsfile runner

Core of solution is *Jenkinsfile Runner*. It allows execution of *Jenkins pipelines* in a container without running *Jenkins master*. With addition of plugins, it is possible to preconfigure Jenkins in such a way that it has access to shared libraries which are implicitly loaded.

Default Jenkinsfile which loads Jenkinsfile from repository is present. This allows pipeline to do some default action, such as if project has file structure according to predefined format, default pipeline is executed.

Request to execute pipeline with its definition is send to *API server* and then pipeline is executed in *Docker swarm*.

7.2.1.6 Jenkins Shared Library

Jenkins *shared library* serves as a main integrating component between *Jenkinsfile Runner* and *Docker Swarm*. It implements new step for existing *Jenkins pipeline* which allows to define *tasks* that are then executed in *Docker Swarm*.

7.2.2 Topology

Reverse proxy is the only input to whole solution. *API server* is tightly coupled with *Swarm manager* as it forwards commands to it.

Jenkinsfile runner and *Task executor* run on *Swarm workers* as stacks deployed to Swarm.

Task executor executes containers on *Swarm worker* outside the scope of *Docker Swarm* orchestration.

This topology is described in figure 7.1.

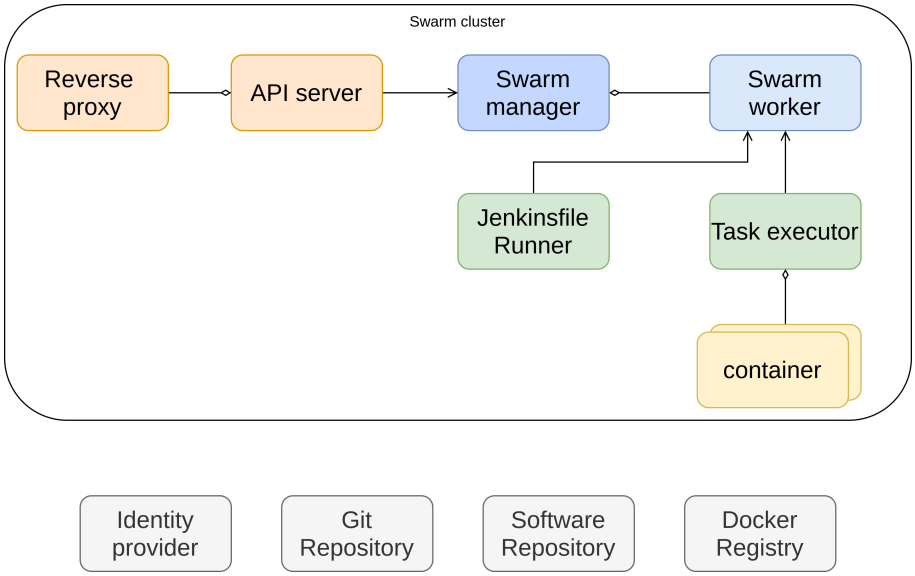

Figure 7.1: Topology of proposed solution

7.2.2.1 Running pipelines

To execute *pipeline*, request to *API server* is made. This request contains parameters for *Jenkinsfile runner* and declaration of pipeline. *Reverse proxy* authenticates this request and *API server* authorizes it. Request is then forwarded to *Swarm manager*. Swarm manager deploys *Jenkinsfile runner* to certain swarm node.

Jenkinsfile runner loads *shared library* and pipeline definition. Pipeline definition contains *tasks* declaration and order, in which they are executed.

Jenkinsfile runner parses task declaration thanks to *Shared library* and sends requests to *Reverse proxy* to execute *Task executor* with definition of task. *Task executor* parses this definition and starts containers accordingly.

While *task* is running, *Jenkinsfile runner* asks for a status of *task* until task is completed or failed.

When *Task executor* finishes, *Jenkinsfile runner* asks *API server* for a status and logs from task executor and sends a request to delete *Task executor*. Jenkinsfile runner executes all tasks in same manner until pipeline is finished.

To help illustrate this flow, figure 7.2 is available. In this figure, flow of *Task executor* deployment is not described as it is analogous to deployment of *Jenkinsfile runner*.

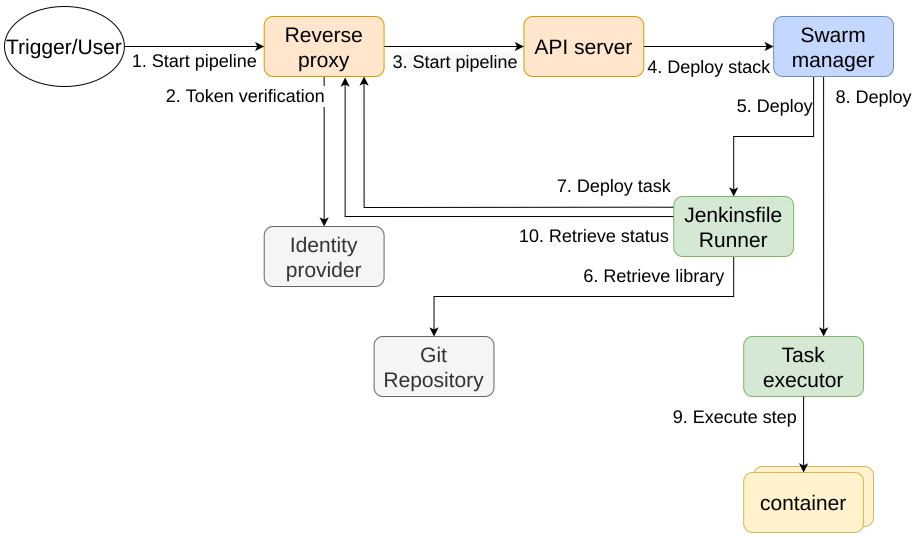

Figure 7.2: Execution of pipeline task

7.2.3 Architecture principles

7.2.3.1 Credentials

Pipelines themselves often need to access some resources. These resources such as software repository, to upload docker image; zip file for CDN or package such as npm or java archive. Also pipeline might need to access another git repository or any arbitrary resource.

Access to credentials used in pipelines is ensured by leveraging *docker secrets*. User can create credentials bound to project by calling *API server* and use them in pipeline via *Jenkinsfile* declaration.

To transfer credentials to tasks runtime, *docker secret* is assigned to *Task executor* stack. Credential is accessible by *Task executor* via temporary file

system mounted to its container. *Task executor* also has additional temporary file system mounted to container. This container is used to share secrets among *Task executor* and its *steps*, which are containers executed by it. As this volume is temporary, contents of it exist only during *Task executor*'s runtime, therefore on completion or failure of a task, credentials are removed.

This approach is better than creating standard docker volume as it would leave credentials accessible as a volume even after *Task executor* finishes its execution. Also it is much more fitter to use than environmental variables as many tools and programs dump environment variables on failed execution.

Whole concept is described in figure 7.3.

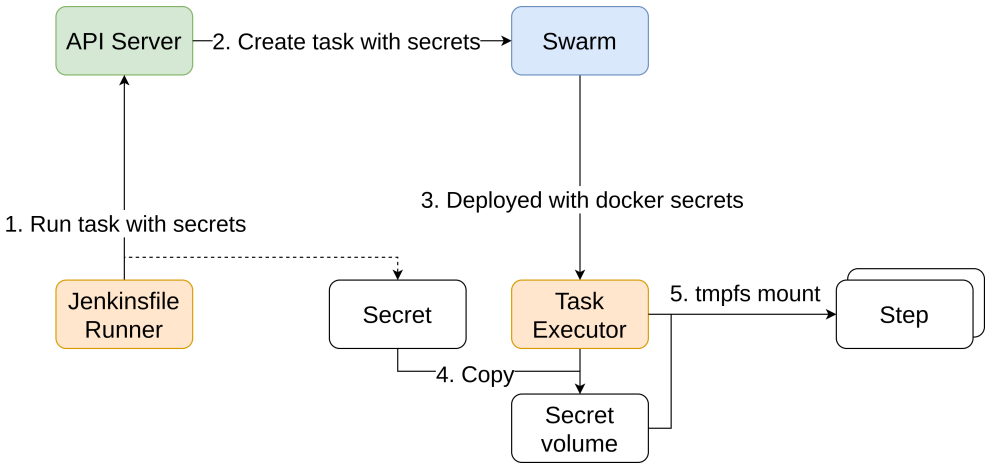

Figure 7.3: Mounting secrets to non-service containers

7.2.3.2 Access control

Authentication and authorization is linked to claims from *OAuth2.0 JSON web token(JWT)* [61]. It is an internet standard that defines a way to securely transmit data among parties[62]. JSON web tokens are signed by asymmetric key. It contains header, payload and signature encoded in *base64* encoding separated by dots('.'). In this usage, header contains algorithm used to sign this token, payload contains claims issued by Identity provider such as subject, to whom token was issued, who issued the token, time of issue, expiration time, etc.

In this case, payload also contains claim to resources to which subject has access rights. This token is provided by *Identity Provider*. It is authenticated by *Reverse proxy* and request for resource is forwarded to *API server* with claims from *JWT* in HTTP headers.

API server authorizes request for resource by parsing it and checking claims against resource. If successful, request passes and in case of failure, request is denied.

7.2.3.3 Self-serviceability

Self-serviceability is based on leveraging *docker secrets* and *Pipeline*'s serverless execution with access rights derived from claims provided in access token from Identity provider.

To make changes to pipeline, such as updating building tools, only change required is to change specification of a pipeline as its modular design fully leverages container and their interchangeability.

Also to create secrets which serve as credentials to access external resources user is required to have access rights to *API server*. This means that credentials does not need to be declared by system administrator.

7.2.3.4 Multitenancy

To separate tenants, claims to resources (or namespaces) must be present in access token. Resources separation and authorization to them must be done on API server as *Docker swarm* itself does not distinguish multiple tenants in cluster. *API server* ensures that only user which has corresponding namespaces present in claims has access to requested resources.

Chapter 8

Prototype implementation

8.1 Pipeline

Pipeline execution engine, *Jenkinsfile runner*, is named *Pipeline* in implementation to distinguish between the two, as implementation used in this solution has additional plugins installed, static but parametrized configuration and predefined *Jenkinsfile*.

8.1.1 Plugins

Workflow aggregator is a collection of plugins implementing Jenkins pipeline. It provides dependencies for individual plugin components of Pipeline implementation in Jenkins.

Configuration as Code plugin is used by Jenkins to load configuration file, which declares basic credentials necessary for proper functionality.

Also additional plugins required as dependency by plugins mentioned above are installed.

8.1.2 Configuration

Configuration provided to *Configuration as a Code* plugin contains library definition and its credentials declaration.

Definition of credentials itself is done via *API server* with help of *Docker secrets*. Reference to defined credentials is then passed to *Pipeline* as an environment variable.

Configuration itself is written in YAML data-serialization language as required by *Configuration as a Code* plugin.

8.1.3 Docker image

Docker image is created from *Pipeline* with all plugins installed. This image is based on *jenkins4eval/jenkinsfile-runner*, which is base image image of

Jenkinsfile Runner [63]. Additional plugins are installed using *Plugin Installation Manager Tool* available from github.com [64]. Also, default *Jenkinsfile* is present in image, which retrieves git repository from which pipeline is run and loads its Jenkinsfile.

8.1.4 Compose file

Compose file to execute pipeline is provided. It defines environment variables used in *pipeline* such as *API server* URL, namespace to which tasks are deployed, username by which tasks are deployed and URL, where this user can retrieve authorization token.

Restart policy is also set to none as other policies would result in indefinite execution of a pipeline.

8.2 Jenkins shared library

Shared library is accessible by *Pipeline* as a git repository.

Shared library itself is written in Groovy as one class. It provides one step, which can be written in Jenkinsfile. This step can be used in *Declarative* or *Scripted pipeline*.

This step contains definition and declaration of task's steps and its metadata. Steps are docker images and arguments that are passed to its execution. Its metadata can be workspaces, environment variables, credentials or working directory, which are all shared among containers.

Class, this library provides, requests *API server* to deploy *task executor* stack with declaration of task. Then it waits for completion of its execution, retrieves execution logs and requests *API server* to delete this stack.

8.3 Task executor

Task executor is implemented as Node.js application, which prepares run commands for docker binary, according to metadata it retrieves. Then it executes them in given order.

Declaration of task is retrieved from environmental variable which is then parsed as a JSON.

To prevent users from injecting malicious code to steps arguments using single(') or double(") quotes or semicolons(;), execution is not done through shell but arguments are passed directly to docker binary.

8.3.1 Docker image

Docker image with built application is created. This image is based on `node` image which has pre-installed binaries for execution of Node.js.

Also *docker binary* is additionally installed as it is required for spawning docker containers.

8.3.2 Compose file

Compose file is also created for Task executor as it is required by *API server*.

It has mount to docker socket on underlying host to ensure that docker containers can be executed. Temporary file system volume is declared for secrets to be stored in and environment variables are passed to it.

Also restart policy must be set to *none* same as in *Pipeline*, otherwise it would lead to endless execution of task.

8.4 API server

API server is written in Node.js technology using *Express* web framework [65]. It provides API for interaction with *Docker Swarm*. Most of commands executed on back end are executed by *docker* binary. Binary provides interface for *docker stacks*.

Before calling *docker* binary, compose file is parsed by *API server* to ensure services use only resources, which are accessible by user. These claims are provided as headers by reverse proxy. If header is appended by `_trusted` suffix, validation is skipped.

8.4.1 API

API is designed as REST based. This means that communication is based on HTTP, is stateless and resources are part of HTTP paths. As this is proof of concept, it does not fully satisfy CRUD (create, read, update, delete) operations.

Every path starts with `/:namespace` identifier, which describes namespace or a project, for which the requests are intended.

Following identifier is a type of resource, which is requested. It can be one of `/stack` or `/secret`. Overview of paths is described in table 8.1.

8.4.2 Docker image

This image is almost identical to *Task executor* image, with only difference are dependencies and source code of an *API server*.

8.4.3 Compose file

Compose file of *API server* and *Reverse proxy* is also available as means of convenient installation on *Docker Swarm*. Services are setup in a way to run on every *Docker swarm manager* as *API Server* communicates with it.

Table 8.1: Overview of API Server's API

Method	Path	Data	Description
GET	/*:ns*/secret	No body.	Returns secrets associated with namespace *:ns*.
POST	/*:ns*/secret/create	Name, base64 encoded secret.	Creates secret associated to namespace *:ns*.
GET	/*:ns*/stack	No body.	Returns stacks associated with namespace *:ns*.
POST	/*:ns*/stack	Name, compose file, parameters for compose file.	Creates stack associated to namespace *:ns*.
DELETE	/*:ns*/stack/:stId	No body.	Deletes stack identified by *:stId*
GET	/*:ns*/stack/*:stId*/services	No body.	Returns services associated with stack identified by *:stId*
GET	/*:ns*/stack/*:stId*/services/*:serviceId*/status	No body.	Returns service status identified by *:serviceId* associated with stack identified by *:stId*
GET	/*:ns*/stack/*:stId*/services/*:serviceId*/logs	No body.	Returns service logs identified by *:serviceId* associated with stack identified by *:sId*

8.5 Apache Reverse Proxy

Reverse proxy is implemented using *Apache HTTP server*. Requests are authenticated using *mod_auth_openidc*, where Apache validates *JWT* signature on URL provided in configuration file. Configuration parameters such as *API Server*'s host and port, and metadata of Identity provider are passed as environment variables.

8.5.1 Docker image

Docker image is also created for reverse proxy. This image has installed *mod_auth_openidc* authorization module with its dependencies and proxy configuration.

Chapter 9

Testing

9.1 Testing environment

This chapter describes components used in testing proposed solution and its configuration.

9.1.1 Docker Swarm

Docker swarm used in testing consists of one *Swarm manager* and two *Swarm worker* nodes. No additional configuration is required as everything else is taken care of by *API server*.

9.1.2 Keycloak

As an Identity provider, Keycloak [66] is used. One testing realm is present. This realm has an OpenID endpoint. This realm has also accessible OAuth2.0 token endpoint, where JWTs are issued.

Users created in Keycloak can belong to groups. These groups represent namespaces, or projects, for which user has access rights. To achieve this, additional mapping in client scope is required. This mapping takes users groups and adds them to *namespaces* claim when issuing JWT.

Two clients are created. Clients in Keycloak are entities which request Keycloak to authenticate user. Usually clients are applications and services. Client for Apache reverse proxy is created as well as client for retrieving tokens which are then sent along requests to start pipeline.

9.1.3 Shared library

Shared library resides in private git repository. This repository is accessible by private *SSH* key, which is defined in *Docker Swarm* using *API server*.

9.1.4 Docker Registry

Private docker registry using Nexus repository manager [67] was created for testing this solution. In this repository, docker images of *API server*, *Apache reverse proxy*, *Pipeline* and *Task executor* reside.

9.2 Demonstration project

Basic project is created to demonstrate capabilites of solution as well as to test it. This project contains simple Node.js script which sums up values included in input file. It has input file and output path mandatory arguments; and optional argument for time measurement.

9.2.1 Git repository

Project resides in private git repository accessible by private *SSH* key.

Repository contains two directories, *src* and *test*. Directory *src* contains simple function and *test* directory contains unit test which is executed during build. Repository also contains simple Dockerfile which builds docker image used in test.

It also contains two Jenkinsfiles, written in both, *Declarative* and *Scripted* syntax.

9.2.2 Jenkinsfiles

Jenkinsfile includes three tasks. One task is building docker image from project and other two are parallel test tasks.

Build task consists of five steps. First retrieves git repository which is meant as a workspace for task. Next, dependencies are installed and unit tests are run. Last step is building a docker image. This step is provided by *kaniko* [68], which is used to build docker images without requiring docker daemon.

Other tasks are purely for demonstration purposes. They are run in parallel and consist of running docker image, which was created in previous step over various data.

Chapter 10

Evaluation

This section focuses on evaluation of metrics, defined in chapter 4.8, regarding solution proposed in chapter 7.

10.1 Pipeline definition

Pipeline is defined as a Jenkinsfile. This file can be placed in arbitrary git repository as checkout must be performed manually in task. This enables to define multiple pipelines in one git repository and therefore, pipelines does not have to be bound to one project.

User is required to have basic knowledge of Groovy syntax or Jenkins Declarative pipeline. Basic knowledge of containers is required as only images and its parameters are provided by users. But to extend pipeline steps, medium knowledge is required as steps consist of docker images.

To declare resources used by pipeline, such as SSH private key for retrieving source code or uploading images to docker registry, user must create this resource by sending request to *API server*. This communication must be done via HTTPS and therefore be encrypted.

As every resource such as pipeline or credentials is created in a *namespace*, and valid token is required to access this resource, basic multitenancy is provided. Resource limiting is another factor of multitenancy but is not implemented in prototype.

10.2 Usage of containers

Proposed solution natively uses containers as execution steps of a pipeline. Containers cannot be set up in arbitrary way. This means that user cannot specify raw arguments to docker. Also it does not provide possibilities to execute commands inside containers. Therefore only basic usage of containers is available.

10.3 Access control

User must authenticate against Identity Provider, which will provide token to user. This token is then sent with every request to *API server*, which is sent to Identity provider by *Reverse Proxy* to be verified. After this verification, user is authenticated. User's request is then sent to *API server* along with claims extracted from token.

Namespaces claim must be present to authorize request by *API server*. *API server* checks if every request to run pipeline, or task, use only resources (volumes, secrets) from specified namespace. Also it checks if pipeline declaration does not contain bind mounts or other added capabilities such as load and unload kernel modules, access device or other. Namespaces claims which are appended by `_trusted` suffix are not checked by *API server* for bind mounts as that would leave *Task executor* unusable.

By using namespace claims, proposed solution provides role-based access control.

10.4 Integration

Authentication and authorization claims are supplied only by Identity Provider as *API server* does not handle users. Therefore, it is fully integrated with *OAuth2.0* identity provider.

Any arbitrary git repository can be used for pipeline definition. As credentials used for accessing it are created by users themselves.

Solution runs natively on *Docker Swarm* and leverages its scheduling and secrets.

10.5 Resource requirements

To appropriately estimate resource requirements, several measurements were made. These measurements can be found in enclosed media. *API server* takes from 28 MiB on idle to around 40 MiB spikes while running 10 simultaneous pipelines. *Apache reverse proxy* requires around 40 MiB on average on 10 simultaneous pipeline runs. *Pipeline* requires around 1GiB of memory for execution. *Task executor* allocates around 33 MiB of memory for execution. As it executes docker containers, this value stays approximately the same.

10.6 Scalability

Pipeline executes in serverless execution model, which means that it is running only when request to run a pipeline is made. It is always bound to execution of one pipeline. Thus, it is scalable by its nature and supports dynamic horizontal scaling.

API server requires *Docker Swarm manager* to forward its requests to and is bound to number of managers present in cluster. Docker recommends running maximum of seven managers in one cluster [27].

10.7 Extensibility

Extensibility of this solution is based on task principle described in 7.2.1.1. It executes containers in specified order and this provides great deal of flexibility as tooling does not need to be installed on host executing the task. Docker images can be created according to needs for specific project.

Another means of extensibility are ***Jenkins Libraries***. As core of this solution is *Jenkinsfile Runner* with custom library, it enables to include another libraries as well. Such library can greatly extend functionality of this solution as is demonstrated in library provided in this solution.

Also ***Jenkins plugins*** can be used to extend functionality of this solution. Numerous plugins are available for *Jenkins* [37]. Plugins in conjunction with Jenkins Libraries enable extending proposed solution inside and outside of pipeline execution.

10.8 Summary

Summarized metrics are available in table 10.1.

Table 10.1: Summary of metrics for proposed solution

Metric	**Criteria**		**Proposed solution evaluation**
Declaration of pipeline	Pipeline as a code		Yes
	User knowledge		Basic
	Declaration format		Jenkinsfile, Groovy script
	Self serviceability		Yes
Usage of containers	Basic		Yes
	Medium		No
	Advanced		No
Access control	Attribute-based access control		No
	Role-based access control		Yes
Integration	Authentication		Yes
	Authorization		Yes
	External git repository		Yes
	Docker Swarm		Yes
Scalability	Static horizontal		Yes
	Dynamic horizontal		Yes
Extensibility	Options	Inside pipeline	Yes
		Outside pipeline	Yes
		Tooling	Yes
	Means	API	Yes
		Plugins	Yes

Conclusion

Objective of this thesis was to analyze requirements, prepare metrics and to design and implement proof-of-concept of a solution for continuous integration and delivery pipeline execution. All of those objectives are met.

Requirements and metrics are prepared in chapter 4. For design of the proposed solution, technologies such as *Jenkins*, *GitLab CI/CD* and Tekton were evaluated in research chapter 5. Research has shown that none of these technologies alone are able to cover all requirements required for the target solution. From architecture point of view, the Tekton technology was the most feasible option but because it cannot be used on the *Docker Swarm* environment, it could not be used directly for proposed solution.

An alternative solution was designed in chapter 7, based on the execution principles of *Tekton* and technology provided by *Jenkins*. This proposal was successfully verified on proof-of-concept implementation 8.

This solution enables serverless execution model which is inherently horizontally scalable. Individual parts of pipeline consist of containers, which provide great modularity. That means that any part of pipeline can be changed by switching containers responsible for its execution. Also direct leverage of containers enables usage of containers which might not have user shell available. This is mostly advantageous for minimalist containers which do not come prepackaged with additional binaries.

Access control is secured by direct integration with *OAuth2.0* Identity provider, where every request must be authenticated by it and then is authorized by proposed prototype. Also basic form of multitenancy is provided by claims from *OAuth2.0* Identity provider, which enable sharing of resources by multiple organizational units.

Aside from solution proposed, content of this thesis can also serve as a research of technologies for some, who might have similar requirements for an execution engine for CI/CD pipelines. *API server* (from chapter Implementation 8.4) can be used as basis for solution for deploying applications to *Docker Swarm* with access control provided by Identity provider and without

the need of giving access to underlying infrastructure. Also *Task executor* serves as an example of parsed execution of set of docker containers without directly providing access to docker daemon.

Use of compose files enables straightforward deployment of services inside arbitrary *Docker Swarm* without any installation requirements to cluster. Also compose files enable simple migration to other container orchestration engines such as *Kubernetes* in a future.

Bibliography

[1] Crosby, M. *What is containerd ? [online].* Docker Inc., [cit. 2020-04-19]. Available from: `https://www.docker.com/blog/what-is-containerd-runtime`

[2] Docker Inc. *About storage drivers [online].* [cit. 2020-04-16]. Available from: `https://docs.docker.com/storage/storagedriver/`

[3] Vincent Driessen. *A successful Git branching model.* Available from: `https://nvie.com/posts/a-successful-git-branching-model`

[4] Davis, J.; Daniels, R. *Effective DevOps: building a culture of collaboration, affinity, and tooling at scale.* " O'Reilly Media, Inc.", 2016.

[5] Synopsys, Inc. *Compare Repositories [online].* [cit. 2020-05-10]. Available from: `https://www.openhub.net/repositories/compare`

[6] Chacon, S.; Straub, B. Pro Git 2nd ed. 2014 Edition.

[7] Sonatype Inc. *Functions of a Repository Manager [online].* [cit. 2020-04-26]. Available from: `https://help.sonatype.com/learning/repository-manager-3/repository-management-basics/lesson-1%3A-functions-of-a-repository-manager`

[8] Rossel, S. *Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment.* Packt Publishing Ltd, 2017.

[9] Humble, J.; Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education, 2010.

[10] The Spinnaker Authors. *Concepts [online].* [cit. 2020-05-27]. Available from: `https://www.spinnaker.io/concepts/`

[11] Red Hat, Inc. *What is virtualization? [online]*. [cit. 2020-04-09]. Available from: https://opensource.com/resources/virtualization

[12] Open Virtualization Alliance. *Kernel Virtual Machine [online]*. [cit. 2020-04-09]. Available from: https://www.linux-kvm.org/page/Main_Page

[13] Mao, M.; Humphrey, M. A performance study on the vm startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, IEEE, 2012, pp. 423–430.

[14] Oracle Inc. *Oracle Solaris Zones Introduction [online]*. [cit. 2020-04-13]. Available from: https://docs.oracle.com/cd/E36784_01/html/E36848/zones.intro-1.html

[15] The FreeBSD Project. *BSD System Manager's Manual [online]*. [cit. 2020-04-13]. Available from: https://www.freebsd.org/cgi/man.cgi?query=jail&format=html

[16] Docker Inc. *What is a Container? [online]*. [cit. 2020-04-15]. Available from: https://www.docker.com/resources/what-container

[17] Pahl, . C. Containerization and the paas cloud. *IEEE Cloud Computing*, volume 2, no. 3, 2015: pp. 24–31.

[18] Rosen, R. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, volume 186, 2013.

[19] The Linux Foundation. *Open Container initiative [online]*. [cit. 2020-04-19]. Available from: https://www.opencontainers.org/

[20] moby. *Docker Image Specification v1.0.0 [online]*. [cit. 2020-04-16]. Available from: https://github.com/moby/moby/blob/master/image/spec/v1.md

[21] Docker Inc. *Docker run reference [online]*. [cit. 2020-05-20]. Available from: https://docs.docker.com/engine/reference/run/

[22] Docker Inc. *Glossary [online]*. [cit. 2020-04-19]. Available from: https://docs.docker.com/glossary/

[23] Docker Inc. *Distribution [online]*. [cit. 2020-04-19]. Available from: https://github.com/docker/distribution

[24] Sonatype Inc. *Docker Registry [online]*. [cit. 2020-04-19]. Available from: https://help.sonatype.com/repomanager3/formats/docker-registry

[25] RedHat Inc. *Project Quay [online]*. [cit. 2020-04-19]. Available from: https://www.projectquay.io/

[26] Eldridge, I. *What Is Container Orchestration? [online].* New Relic Inc., [cit. 2020-04-19]. Available from: https://blog.newrelic.com/engineering/container-orchestration-explained

[27] Docker Inc. *How nodes work [online].* [cit. 2020-04-19]. Available from: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes

[28] Portworx Inc. and Aqua Security Software Inc. *2019 Container Adoption Survey [online].* [cit. 2020-05-01]. Available from: https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf

[29] Baldini, I.; Castro, P.; et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, Springer, 2017.

[30] Gartner. *Multitenancy.* [cit. 2020-03-04]. Available from: https://www.gartner.com/en/information-technology/glossary/multitenancy

[31] Opensource.org. *Compare Repositories [online].* [cit. 2020-05-10]. Available from: https://opensource.org/licenses

[32] CD Foundation. *Continuous Delivery Landscape [online].* [cit. 2020-05-27]. Available from: https://landscape.cd.foundation/

[33] Jenkins. *Jenkins User Documentation [online].* [cit. 2020-04-23]. Available from: https://jenkins.io/doc/

[34] Kawaguchi, K. *MIT License[online].* Jenkins, [cit. 2020-04-26]. Available from: https://www.jenkins.io/license/

[35] jenkinsci. *Jenkins infra-statistics [online].* [cit. 2020-04-25]. Available from: https://stats.jenkins.io/jenkins-stats/svg/svgs.html

[36] Jenkins. *Jenkins Handbook [online].* [cit. 2020-04-25]. Available from: https://www.jenkins.io/doc/book/

[37] Jenkins. *Plugins Index [online].* [cit. 2020-04-24]. Available from: https://plugins.jenkins.io/

[38] Gleske, S. *GitHub Authentication [online].* [cit. 2020-05-22]. Available from: https://plugins.jenkins.io/github-oauth/

[39] Michael Bischoff, S. A. *OpenId Connect Authentication [online].* [cit. 2020-05-22]. Available from: https://plugins.jenkins.io/oic-auth/

[40] Yasuyuki, I. *Authorize Project [online].* [cit. 2020-05-22]. Available from: https://plugins.jenkins.io/authorize-project/~

[41] Thomas Maurel, O. N., Romain Seguy. *Role-based Authorization Strategy [online].* [cit. 2020-05-22]. Available from: https://plugins.jenkins.io/role-strategy/

[42] danielbeck. *Matrix Authorization Strategy [online].* [cit. 2020-05-22]. Available from: https://plugins.jenkins.io/matrix-auth/

[43] Surya Gaddipati, R. B. *Docker Swarm [online].* [cit. 2020-05-22]. Available from: https://plugins.jenkins.io/docker-swarm/

[44] GitLab Inc. *GitLab Docs [online].* [cit. 2020-05-22]. Available from: https://docs.gitlab.com/ce

[45] Mulvany, A. *Docker Swarm integration for GitLab-CI [online].* [cit. 2020-05-23]. Available from: https://gitlab.com/gitlab-org/gitlab-runner/-/issues/2485

[46] GitLab Inc. *Runner Docs [online].* [cit. 2020-05-22]. Available from: https://docs.gitlab.com/runner

[47] The Kubernetes Authors. *Custom Resources [online].* [cit. 2020-05-22]. Available from: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[48] The Kubernetes Authors. *Reference [online].* [cit. 2020-05-24]. Available from: https://kubernetes.io/docs/reference/

[49] The Kubernetes Authors. *Installing Kubernetes [online].* [cit. 2020-05-24]. Available from: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

[50] The Tekton Authors. *Design PipelineResource extensibility [online].* [cit. 2020-05-24]. Available from: https://github.com/tektoncd/pipeline/issues/238

[51] The Tekton Authors. *Design alternative Task Implementations inside a pipeline [online].* [cit. 2020-05-24]. Available from: https://github.com/tektoncd/pipeline/issues/215

[52] Jenkins. *Pipeline as Code[online].* [cit. 2020-04-27]. Available from: https://www.jenkins.io/doc/book/pipeline-as-code/

[53] Google Inc. *Google Trends - docker swarm [online].* [cit. 2020-05-27]. Available from: https://trends.google.com/trends/explore?date=today5-y&q=dockerswarm

[54] Golshan, A. *The continuing rise of Kubernetes analysed: Security struggles and lifecycle learnings [online]*. [cit. 2020-05-27]. Available from: https://cloudcomputing-news.net/news/2019/aug/29/continuing-rise-kubernetes-analysed-security-struggles-and-lifecycle-learnings/

[55] Docker Inc. *Manage data in Docker [online]*. [cit. 2020-05-22]. Available from: https://docs.docker.com/storage/

[56] jenkinsci. *Declarative Jenkins Pipelines (Pipeline Model Definition Plugin) [online]*. [cit. 2020-04-25]. Available from: https://github.com/jenkinsci/pipeline-model-definition-plugin/

[57] jenkinsci. *Official Jenkins Docker image [online]*. [cit. 2020-04-26]. Available from: https://github.com/jenkinsci/docker

[58] jenkinsci. *Plugin Installation Manager Tool [online]*. [cit. 2020-04-26]. Available from: https://github.com/jenkinsci/plugin-installation-manager-tool

[59] jenkinsci. *Jenkinsfile Runner [online]*. [cit. 2020-04-26]. Available from: https://github.com/jenkinsci/jenkinsfile-runner

[60] The Kubernetes Authors. *Translate a Docker Compose File to Kubernetes Resources [online]*. [cit. 2020-05-22]. Available from: https://kubernetes.io/docs/tasks/configure-pod-container/translate-compose-kubernetes/

[61] M. Jones, C. M., B. Campbell. *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants [online]*. [cit. 2020-05-22]. Available from: https://tools.ietf.org/html/rfc7523

[62] oauth.com. *Introduction to JSON Web Tokens [online]*. [cit. 2020-05-20]. Available from: https://jwt.io/introduction/

[63] jenkinsci. *Jenkinsfile Runner [online]*. [cit. 2020-05-22]. Available from: https://github.com/jenkinsci/jenkinsfile-runner

[64] jenkinsci. *Plugin Installation Manager Tool [online]*. [cit. 2020-05-22]. Available from: https://github.com/jenkinsci/plugin-installation-manager-tool

[65] TJ Holowaychuk, StrongLoop, et al. *Express [online]*. [cit. 2020-05-22]. Available from: https://expressjs.com

[66] JBoss. *Keycloak [online]*. [cit. 2020-05-22]. Available from: https://www.keycloak.org/

[67] Sonatype Inc. *Nexus Repository OSS [online]*. [cit. 2020-05-22]. Available from: `https://www.sonatype.com/nexus-repository-oss`

[68] GoogleContainerTools. *kaniko - Build Images In Kubernetes [online]*. [cit. 2020-05-22]. Available from: `https://github.com/GoogleContainerTools/kaniko`

Appendix A

Acronyms

ABAC Attribute-based access control

API Application programming interface

CI Continuous integration

CD Continuous delivery

CDN Content delivery network

CLI Command line interface

CRUD Create, read, update, delete

CPU Central processing unit

DSL Domain-specific language

DRY Do not Repeat Yourself

GPU Graphical processing unit

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON JavaScript object notation

JWT JSON Web Token

OCI Open container initiative

OIDC OpenID Connect

OS Operating system

RAM Random Access Memory

RBAC Role-based access control

REST Representational state transfer

SSH Secure Shell

SSL Secure Socket Layer

TLS Transport Layer Security

URL Uniform Resource Locator

URI Uniform Resource Identifier

VCS Version control system

XML Extensible markup language

YAML YAML Ain't Markup Language

Appendix B

Contents of enclosed media

- `readme.txt` ..... the file with media contents description
- `src` ..... the directory of source codes
  - `practical` ..... practical part of thesis
    - `main`
      - `api_server` ..... implementation sources of API Server
      - `reverse_proxy` ..... implementation sources of Reverse Proxy
      - `pipeline` ..... implementation sources of Pipeline
      - `task_executor` ..... implementation sources of Task Executor
    - `task_library` ..... implementation sources of Task Library
    - `demonstration_project` ..... demonstration git repository
    - `measurements` ..... measurements of solutions resources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `DP_Le_Henrich_2020.pdf` ..... the thesis text in PDF format