Exploring CI/CD Pipelines for Cloud Infrastructure Deployment

Can one increase efficiency through amalgamation?

Andreas Faanes Slagsvold

Master Thesis
in
Applied Computer and Information Technology (ACIT)
30 credits

Department of Computer Science
Faculty of Technology, Art and Design

OSLO METROPOLITAN UNIVERSITY

Spring 2023

Exploring CI/CD Pipelines for Cloud Infrastructure Deployment

Can one increase efficiency through amalgamation?

Andreas Faanes Slagsvold

Abstract

Cloud computing has seen a massive surge in popularity over the last decade, and these days it is the go-to solution for many companies and private actors alike when it comes to how they chose to run their programs, systems or storage. With the ever-increased emphasis on scaling and efficiency that this popularity causes, researchers and administrators within the industry have begun turning their attention to DevOps principles, particularly the use of CI/CD pipelines to handle the demand, but such implementations are still relatively uncharted territory. This paper presents an overview of such CI/CD pipelines, the core concept and how such implementations have been attempted, before presenting an experimental pipeline implementation that could potentially contribute to the building of future cloud infrastructure deployments.

Keywords: Cloud Computing, Continuous Integration, Continuous Deployment, CI/CD Pipelines, Orchestration Tools

Acknowledgments

I would like to offer my sincere thanks to my supervisor, Syeda Taiyeba Haroon, without whom this paper would have been impossible. Her counseling and guidance has been excellent through and through, despite her having to face sickness for a good portion of the time frame. For someone being a supervisor for the first time, I do not think one could have done a better job.

I would also like to thank my parents, who have supported me throughout the project and pushed me to keep my motivation and focus up through the whole project time frame.

Lastly, I would like to thank my friends and family who have taken time out of their day to help me with control reading, letting me know where I can improve the text and generally solidifying my work. Their input has been invaluable and contributed greatly to the thesis getting to this point.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, the use of cloud technologies has seen a surge in popularity, and the topic of cloud storage and utilization has been a dominant topic in corporate strategies for a while. At an individual level, cloud technologies meet you almost wherever you turn, whether it is for storing images on your phone, to logging on to your favourite streaming service when you come home from work, some aspect related to the cloud are there. Another increasingly prevalent technology that has come into fruition as a technological advancement, however, is the concept of continuous integration and deployment, or CI/CD for short. While CD is often referred to both as continuous deployment and continuous delivery [46], we have chosen the deployment variant given the context of this project. CI is a concept which involves keeping the entire application code in a common repository and ensuring that the latest code, through scripts, is picked from the repository, integrated with the existing code and run through a series of test cases whenever a developer deploys code into the repository. CD expands on this by ensuring that integral aspects such as new features and bug fixes can be deployed straight to the live servers running the application when needed [52]. These days, companies use the CI/CD pipelines more and more to efficiently deliver software, because it makes the process of delivery much easier. This originally stemmed from CI's ability to merge and integrate development work very frequently, improving software quality and increasing productivity [48]. This ability has been further enhanced with the emergence of CD, adding even further potential to the technology.

Additionally, CI/CD pipelines make it easier for teams and sections within or across companies to work together in a good way, by making it easier to set up a good DevOps culture within the workplace. DevOps is a relatively new concept that has become a very common developmental approach in recent years. In a 2016 study, it was defined as a "development methodology aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices [27]. Taking a DevOps approach often involves merging the two branches into one team, where the engineers contrary to more traditional approach work across the entire life cycle of the application, developing skills that does not limit themselves to one function [1]. In Figure 1.1 this merge is displayed through an illustration of the DevOps life cycle, where development continuously runs throughout the different stages of the cycle in a constant process. The results of using this approach have seen a lot of benefits that are deemed quite valuable in today's fast-paced environment. Stillwell and Coutinho [54] for example addressed how the implementation of such a DevOps approach in the EU FP7 project HARNESS led to the development teams being able to operate more autonomously, while at the same time being more open to new changes and more frequent updates. The communication overhead also decreased, further adding to the efficiency of deployment.

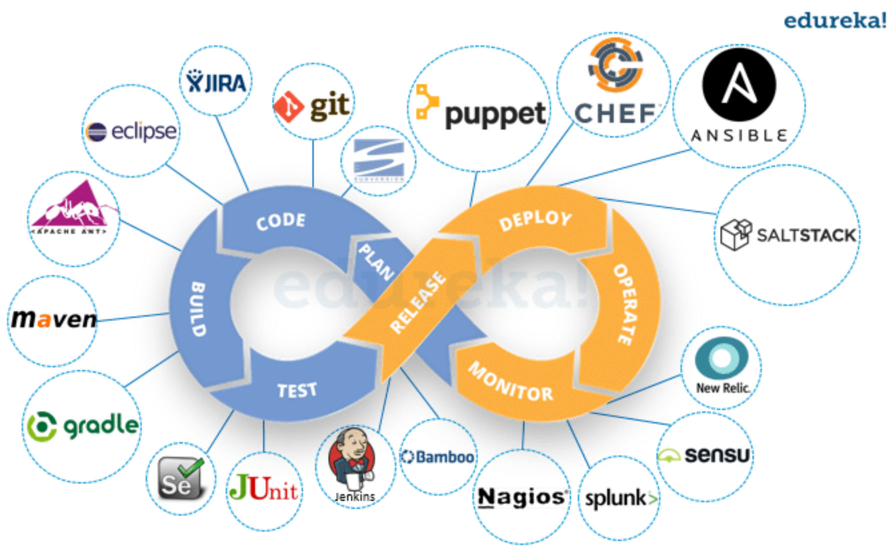


Figure 1.1: The DevOps life cycle [15]

In present time, because of how companies are scaling their clouds, a disadvantage of the current practices is that one sees increased complexity of deploying and maintaining the cloud structures. Since cloud technologies has seen such a rapid increase in popularity and demand, one could argue that some aspects have been unable to keep up with the massive pace. With the emergence of new cloud tools such as configuration management, great progress has been made. Even so, today's tools do not, however, contribute in deploying and testing cloud deployments automatically. Instead, they focus primarily on letting companies provision and monitor the infrastructure they use such as puppet, chef, ansible for configuration management, salt stack for provisioning etc as illustrated in figure 1.1. Through these tools, then, it is easy to discover when something is out of the ordinary within the system, but the tools used does not offer much in terms of automatically testing why these uncommon occurrences happen. Instead, one is required to manually test and investigate the issue if no other solution is present, demanding a lot of resources in terms of human capital and time.

While some companies do use CI/CD pipelines to deploy clouds today, the concept is still so fresh that it is far from common practice, and there are no automated tools dedicated to cloud deployment testing available currently. With the implementation of CI/CD pipelines for clouds, companies would be enabled to address these challenges of automatically testing cloud deployments, which in turn would help discover bugs or mis-configurations earlier in the deployment process, thus increasing efficiency. Since CI/CD pipelines work so well for efficient software testing and delivery, such an implementation could hold a big potential to having more efficient cloud deployments, as it addresses some of the core issues with the current deployment standards. If successful, one could even look at the possibilities of developing tailored CI/CD pipeline tools specifically for cloud deployments in the future.

1.1 Problem Statement

Our problem statement aims to enhance the way we look at the complexity of cloud deployment and explore ways one could implement continuous integration and testing of cloud deployments.

The problem statement we are going with for our work is:

Explore the use of continuous integration and deployment of cloud infrastructure, with regard to cloud operations.

"Continuous integration and deployment" are in this case a term used for describing a practice within the software development industry. Since the practice involves rapid integrations and merges of development with frequent changes and updates, the terminology in our case would involve a system of rapid development and deployment of the cloud structure in question. This is because the goal of the project involves looking into a system that can utilize automatic testing and deployment as a tool for extending the efficiency of the cloud. With continuous integration, you get shorter and more frequent release cycles through automated building and testing, which fits perfectly for our idea. This is further enhanced by continuous deployment, which, as the name indicates, also involves automatically deploying changes into the production environment. Figure 1.2 shows an illustration of a typical CI/CD pipeline structure with the various stages. This image was originally created as a CI/CD pipeline template, making it a solid fit for illustrating what our pipeline could potentially look like. It depicts the different stages of development in a typical pipeline, starting with the developer committing some change to the source code, moving through automation and containerization and going all the way to production.

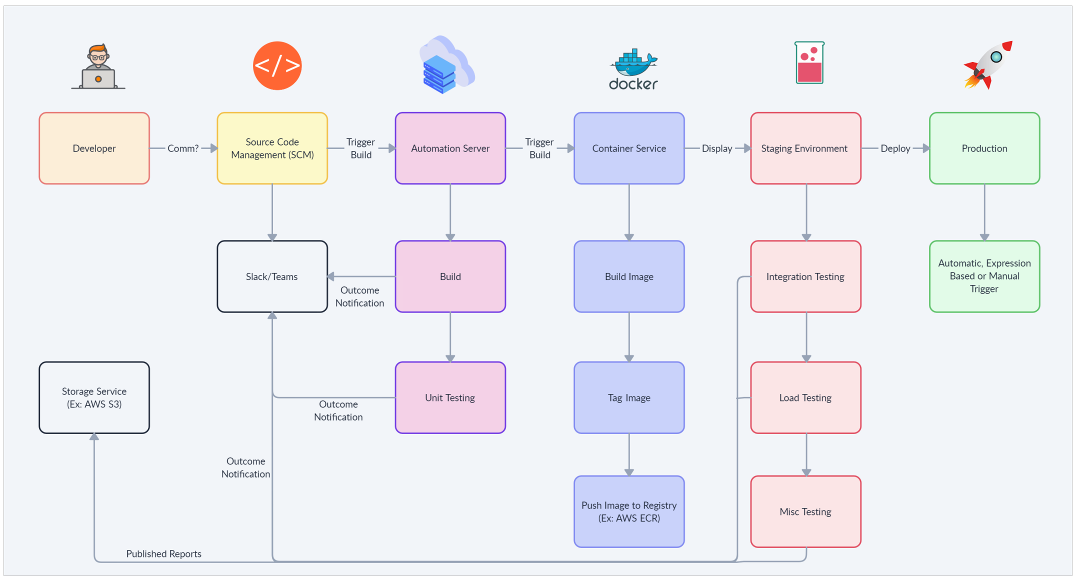


Figure 1.2: An example of a CI/CD pipeline structure [12]

“Cloud infrastructure” is used to describe the different aspects required for cloud computing to work, from hardware to network resources. Since this project will rely on a cloud deployment environment of some kind, one of the core components of the cloud infrastructure that will be looked at here are virtual machines, as well as network resources and storage.

Finally, “cloud operations” is the summarizing of different actions required to manage IT services running in a cloud environment in a satisfactory manner. Within this category one can find another key term associated with cloud technologies today, “configuration management”. The tools used within this umbrella term can be viewed as tools used by cloud structures to monitor changes within the structure itself automatically. Since an application or structure running on the cloud usually involves multiple running parts to make up the complete package, such tools are important to give an overview of every component, and whether or not it is running as intended. A prominent example of such is Puppet, which makes it easy for an administrator to check that all the virtual machines are running smoothly, for example. In this project, if successful, such tools will work together with the CI/CD pipeline to ensure a solid foundation for detection of errors or irregularities within the infrastructure. Another typical term associated with cloud operations is "orchestration tools", tools that are used to automate tasks and processes to ensure a more efficient workflow. Examples of this would be Terraform and Kubernetes, something that will be further addressed in the following sections. For this project, once the prototype has been established, some tests will be required to run on an environment running some cloud operations, with and without the CI/CD pipeline implementation, to study the effect this has on the efficiency and flexibility of the deployments.

1.2 Thesis outline

The following section presents an outline of the structure of the remainder of the thesis.

Chapter 2 elaborates on the considerations leading up to this research, the relevant technologies existing on the current market and past research that has been committed on the field and on the various technologies explained.

Chapter 3 explains how our problem statement was put into a relevant context in terms of how we could approach the task at hand. It details the different phases of the project, with our motives behind our structuring, what we hope to find and how we initially planned on going about the prototype creation explained in later sections.

Chapter 4 gives detailed accounts of our results in the first phase of the project. Here, we display and explain the various findings from the categorization work with previous research, what we found valuable from these findings and how we take this value into the next phase of the project.

Chapter 5 builds on the results of chapter 4 and explains how this information was put into use, in combination with previous and personal knowledge on the topic, when designing the structure of our prototype. It describes how we decided on the structure of the pipeline, the tools we chose to utilize as well as the outcome of how the prototype functioned at the time of implementation.

Chapter 6 elaborates on the work we have done, which things worked, which things did not and what steps can be taken to circumvent such problems in future endeavours. It also gives comments on our experience with doing this type of project work over an extensive time period, with elements such as time management, restructuring in the face of obstacles etc.

Chapter 7 goes into detail on the obstacles met while working with the project in each phase, where these issues occurred and potential workarounds to ensure progression in the project. The chapter also discusses aspects with the prototype that we would have liked to put further enhancements into, as well as a section for ideas to future work and improvements on our work.

Chapter 8 summarizes the work and findings from the thesis, how our work can be used in connection to the problem domain and whether we reached our goal.

Chapter 2

Background

Picture the following scenario: You are part of a group of developers in a company. Your work consists of working with the development and maintenance of the company application, the pride and joy of the company, who takes great pride in delivering constantly online services to the customers. This has largely been possible after the company moved its operations to the cloud because it has given you and the rest of the team better conditions for working with the application with a DevOps approach. Through continuous integration and delivery pipelines within the cloud, you have been able to make updates and deployments a lot more frequently, which in turn has not only made it easier to deliver the software to the customers but also easier for the team to work together. A win for everyone involved.

Today, however, you are in trouble. Tension within the team is evident, and the stress levels are rapidly increasing. The situation at hand is new to you; since the move to the cloud, things have largely been smooth sailing for the team. Development and updates have been going well, so well, in fact, that the company has seen a drastic increase in customer numbers. This increase has recently led to the company scaling up their cloud to be able to welcome the expanded customer base, a move considered positive by you, as well as everyone else. After all, with the success so far, it should only be expected that following the same structure to a bigger scale would only lead to continued success, right?

Even after the increased scaling, everything has seemed fine. Thus, the problem that has arisen feels like it's come out of nowhere, and the situation feels like it's rapidly getting out of hand. Something is wrong, but not with the application. No, the source of error is within the cloud itself. Through the configuration management system that you implemented when setting up the cloud infrastructure initially, the red indicator sticks out like a sour thumb.

The error indication is unmistakable. It is no doubt that the system is doing its job. The problem, however, is that this is also as far as it goes. From here, the team is largely on its own in finding out exactly what the error is.

As the struggle continues, you realize that in your excitement to upscale, and blinded by the success of the company, an underestimation has been made. The way the company has been scaling the cloud it uses has led to increased complexity in terms of deploying and maintaining the structure, an issue that is common for every cloud infrastructure in today's market and are a constant topic of discussion. However, it is becoming more and more evident that this has been taken too lightly here. Something has gone wrong with the maintenance, and now you are paying the price. You have tried to look through the entire configuration management interface for clues as to what has gone wrong, but without luck. Time is precious, so the team has started the gruelling work of manually checking the different component for the issue. This is a slow and tedious process, but you see no other options; it will only be a matter of time before the operations are halted, which could be a massive detriment for the company.

As frustration and desperation levels reach their apex and the first notifications that customers are struggling with the availability of the application, a team member finds the issue. Quickly, the team manages to correct the maintenance mistake. Things quiet down, services go back to full operationality. Crisis averted for now. However, this was a close call, and as you get ready to pack up for the day, a thought goes through your head; is it possible with a system that could have prevented this from happening? How will you ensure that the company is better equipped in the future?

While the scenario described here is purely theoretical, it still presents a conundrum that is becoming increasingly evident in today's cloud-based landscape, a problem that has grown with the rising popularity of both DevOps culture and the scaling of clouds. A clear trend in today's market is that companies choose to utilise CI/CD pipelines to a bigger and bigger degree to efficiently deliver software, because such pipelines make delivery much easier. As stated by Sikender Mohammad in his 2016 publication, automation in DevOps and DevOps culture, a concept that usually follows CI/CD implementation, tends to increase development speed and precision, while also granting higher consistency and reliable delivery rates. With such pipelines, the software provided or used by the companies run continuously in a whole different matter than was normal in previous technologies, and the benefits of CI/CD also leads to the application quality increasing [37].

The use of automation, to which such pipelines contribute, can be applied to almost any aspect of the development line, not only to software delivery. Automation can also be used to do evaluations of different IaaS cloud options for companies and private actors, for example, by being able to do time- and labor-intensive tasks and thus mitigate much of the effort required to do these tasks manually [51] In a work focused on experimental automation of the cloud dating back to 2013, there is a description of the development of a tool called CloudBench that automates the process of conducting experiments in cloud computing environments. The tool aims to provide researchers and practitioners with a more efficient way of conducting experiments in the cloud, by automating the deployment of cloud resources and the execution of experiments, as well as providing tools for monitoring and analyzing the results. They also demonstrate the usefulness of the tool through a case study, in which they use CloudBench to conduct experiments on two different cloud platforms. A tool like this is a stepping-stone towards automating cloud deployments. This article thus highlights how broad the automation concept can span within the cloud technologies.

Additionally, the emergence of DevOps culture thanks to the CI/CD implementation has also led to many developer teams within or across different companies being able to work together in more efficient ways. This was exemplified through the HARNESS project in 2015, in which an EU FP7 project had software components implemented into the project with a DevOps approach in mind. In the documentation of this experiment, it was concluded that the approach led to the developer teams associated with the project improving their ability to work autonomously, while simultaneously becoming more open-minded when it came to changes and frequent updates. The implementation also led to less communication overhead than before, leading to the researchers concluding that the DevOps approach taken had contributed to a clear improvement of speed and efficiency in terms of reaching the project milestones [54].

Since the development over recent years has led to more and more technology and applications being moved over to the cloud, the demand for cloud servers is ever-increasing. With a larger and larger part of the population also getting access to computers and internet, this also places a strain on the servers and in combination leads to the companies having to increase the scaling of their clouds more and more. While this ensures the access to their applications for the growing customer base, the problem in the initial story also becomes an increased risk; with the rate the companies are scaling their clouds, the complexity of deploying and maintaining these clouds increases. This is illustrated by Geist and Reed who in 2015 discussed how increased scaling leads to strain in the system both in terms of application and system complexity, structuring as well as energy consumption concerns that had to be figured out, and that commercial cloud computing vendors have embraced component failure as a common event [20].

While tools have been developed to try to handle these problems, most of the tools being utilized today, such as configuration management tools, have uses mostly centred around the contents within the clouds. With configuration management, companies have much better options and control when it comes to monitoring the contents of their clouds and can easily discover if something is not running properly or some unwanted occurrence has happened within the system. With the most popular options such as Ansible and Puppet, the nodes are also to some degree maintained by the configuration management through the use of configuration files, letting developers more easily make changes and maintain, in other words manage, their nodes within the system [47] [25]. This is the big contribution of current tools, and why they have made a big improvement on the complexity problem associated with increased scaling. It does not, however, solve the problem completely, as the current tools are mostly centred around monitoring and maintaining the contents within clouds, and not the clouds themselves. While configuration management systems make it easy to detect uncommon occurrences, for example, these systems do not contribute much in terms of automatically testing why these occurrences have happened. Similarly, configuration management tools do not provide much in terms of automatically testing and deploying cloud deployments either, instead ensuring the installation and setup of applications within the cloud mainly.

As previously addressed, the main forte of CI/CD pipelines is providing solid grounds for efficiency in terms of software testing and deployment. Because of this, and the DevOps culture they often bring, utilizing this could enable companies to address the challenges of automatically testing and deploying cloud deployments, through properties like centralized pipelines, full audit trails of previous work and more automation, making it easier to control what is being deployed, when and how it is being deployed etc. [17]. Research on the field has also uncovered possible strategies of implementation of such pipelines, often with great success. In their 2022 research, for example, Donca et al. emphasize the importance of CI in improving software quality and reducing development time and propose a pipeline generator that ensures high availability without any downtime, making scalability easy and reliable as well as having automatic rollbacks and other reactions based on the actions and events caused by changed, thus addressing several of the core topics related to the cloud scalability issue [14].

Another big benefit with CI/CD in cloud deployment automation is that it could enable developers to put their focus on other aspects associated with their work. It has for example been stated that automated pipeline introduction increases the self-provisioning aspect already present within cloud computing, enabling teams to focus on the development of business code or automation enhancement. Such implementations have proven to give repeatability and correctness with every iteration, providing higher velocity and agility while decreasing bugs and breaches, or at the very least ensuring that these are detected earlier in production. These increases also do not affect other important aspects like quality and security [21]. Thus, it is evident that looking into ways of properly implementing such pipelines into cloud deployments, through investigating other research on the field and through some level of prototype creation, could have the potential to give better insight into a still relatively uncharted territory when it comes to cloud technology. This insight could in turn lead to the discovery of strategies for more efficient cloud operations altogether, both through increased efficiency and speed as well as through more efficient use of resources. Because of this, the topic seems a highly relevant one to commit more research on.

2.1 Technologies

The following section describes various technologies that all share the common trait that they have seen a massive increase in usage with the emergence of cloud computing, and many of them are now more or less required for a cloud-based system to withhold today's standards for such systems. This is due to the fact that a cloud system consists of different components that have specific use cases that make them very valuable tools for the systems of which they are a part. An example of this is the aforementioned configuration management concept, which is used for monitoring and management of a system, used because of stability and performance. Similarly, the rest of the technologies mentioned here have tasks and uses which make them invaluable to their respective fields and areas.

2.1.1 Continuous integration/ Continuous delivery

As previously mentioned briefly, continuous integration and continuous delivery are concepts that ensure continuously running software, with several benefits such as higher deployment rates, increased quality and higher reliability being brought from this way of thinking about development [37]. In the world of CI/CD, there are several well-known options that can be used to create and set up workflows containing such pipelines. A pipeline, in the computational sense, can be seen as a continuous process that contributes to software development being driven forward through the use of build, deploy and test-sections. These sections are done automatically through script-files, leading to chance of human error being diminished, as well as ensuring that the software release process is consistent at all times. [45] One of the more cutting edge options that a lot of companies and developers alike elect to go with for their pipeline creations in the modern landscape is GitLab, a tool that we will be showcasing further in this project.

GitLab

Figure 2.1: GitLab has become the go-to CI/CD tool for many [18]

GitLab was released in 2011, today functioning as a code hosting and issue-tracking web platform and continuously increasing in popularity. It is an open-source software owned by GitLab Inc., but with over 2000 different contributors to the software [38]. The core element that a solution of which this project is concerned with requires, was the implementation of a CI/CD pipeline to the rest of the technologies described. It is for this task GitLab is such a well-known tool to handle.

Gitlab utilises the Git version control system. Version control is a concept that lets developers manage and control changes and updates to their code through branching and commits. Commits can be seen as “save points”, where the version control system saves the work the developer does at certain points, making it possible to revert the code back to these states if something should go wrong during development. The branching is a logistical component where the software can be split into different “departments”, so that different developers can make changes without disrupting each other′ work. For larger software companies this is a very beneficial ability, because it allows different developers to work on different services or features at the same time [38].

A known concept with git is that a developer that contributes work to a project needs a copy of said project stored locally on their computer, and since GitLab is built on top of git, the same concept applies. GitLab provides a web interface that makes it easier for users to handle the more intricate aspects of git workflows, as well as recommending a workflow to handle interaction with git for as high a level of productivity, efficiency and ease of use as possible. It offers several useful features like a file browser, branch- and tag viewers, commits graph analysis tools and many others. It is also one of the main standards for version controlling used by developers today (O′Grady, 2018), as well as being well established as compatible with the dominant automation tools in today’s market, such as Terraform [55] [3]. These considerations, along with the large quantity of accessible documentation on the software made this the most suitable option for the implementation of the CI/ CD pipeline.

2.1.2 Cloud Technologies

Cloud computing is a concept that has seen a massive surge in popularity over the last 20 years and is now one of the most dominant computing technologies on the market, with ever more companies and industry leaders moving their businesses over to various cloud solutions. The concept can be seen as the on-demand consumption of compute power, storage, database, applications, and any IT resources through the Internet following a pay-as-you-go price model. Or, in the most basic of terms, a Cloud can be defined as a computer that is located at a different location, that is being accessed through the internet and that is being utilized in some way [44]. In other words, cloud solutions have made resource management and efficiency in terms of scaling and usage more robust and easier for developers and companies alike, and today a plethora of options are available to choose from. One of these options is OpenStack.

OpenStack

OpenStack is a software tool collection used to build and manage public and private clouds launched by NASA and Rackspace Hosting in 2010. It is an open-source platform for computing, but also includes free and open-source software that was released based on the terms of the Apache licence [31]. It provides all utilities needed for end users in particular to launch virtual machines in the form of instances [57]. These can then easily be managed, updated or deleted through the OpenStack interface. It consists of the following components [40] [31]:

Nova, which functions as the main computing engine in the OpenStack infrastructure, enabling the deployment and management of large quantities of instances.

Swift, which is where files and objects within OpenStack are stored. In OpenStack, such storage is done by the system itself, where the developer only needs to refer to an identifier to the file or piece of information needed, enabling increased scaling, as the system is responsible for much of the logistics centred around storage.

Cinder, which is a block storage component, more similar to the classic idea of computers accessing specific disk drive locations. It is commonly used for OpenStack Compute instances.

Neutron, which acts as the networking component of OpenStack, providing the networking capabilities of the infrastructure and making management of IP addresses and networks easier.

Horizon, the dashboard component of OpenStack, easing management and deployment of instances through a graphical web interface.

Keystone, the component responsible for the identity services of OpenStack. It functions by having an extensive list of all users mapped against all accessible services provided by OpenStack for those users.

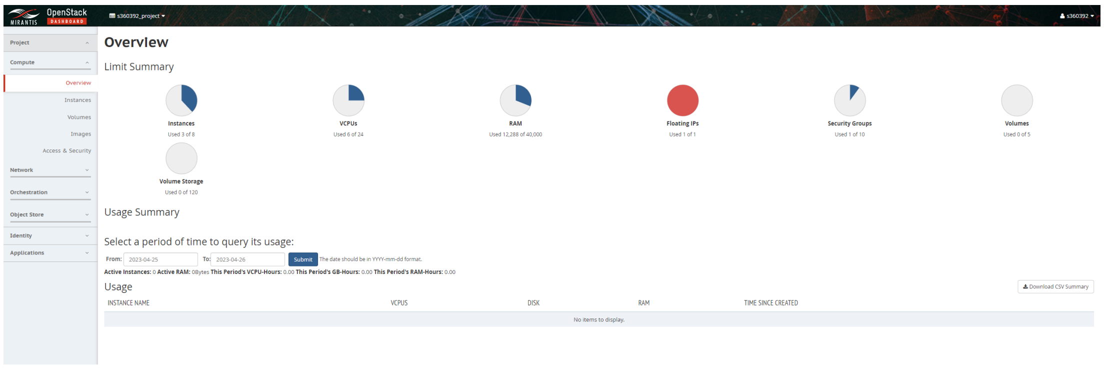
Glance, an image service provider, where images in this particular case are hard disk images that can be used as templates for future virtual machine instances through glance.

Ceilometer, which enables OpenStack to track the resource usage of each user as well as provide billing services through telemetry service provision.

Heat, which acts as the orchestration component of OpenStack, allowing developers to create files containing cloud application requirements, defining the resources required for the application and then storing these files.

Trove, a database as a service component, providing relational and non-relational database engines.

Using cloud solutions like OpenStack gives you some major benefits over other options like for example a local solution running on physical hardware. Utilizing a cloud option, for example, opens up more possibilities in terms of how one can work during a project. Many people also end up deciding to go for OpenStack specifically, because it is a well-established solution on the market, making documentation available in large quantities. It is also, as previously mentioned, an open-software source code solution, and many people find that it is beneficial to make use of this free open-source software as opposed to using potentially even more well-established solutions like for example Amazone AWS. Some of the core features, like the Horizon web interface, also make OpenStack an intuitive and easy to use solution to go for, making it a suitable option for many types of projects.



Overview

Limit Summary

Instances Used 3 of 8 · VCPUs Used 6 of 24 · RAM Used 12,288 of 40,000 · Floating IPs Used 1 of 1 · Security Groups Used 1 of 10 · Volumes Used 0 of 5 · Volume Storage Used 0 of 120

Usage Summary

Select a period of time to query its usage:

From: 2023-04-25 To: 2023-04-26 The date should be in YYYY-mm-dd format.

Active Instances: 0 **Active RAM:** 0Bytes **This Period's VCPU-Hours:** 0.00 **This Period's GB-Hours:** 0.00 **This Period's RAM-Hours:** 0.00

Usage

INSTANCE NAME	VCPUS	DISK	RAM	TIME SINCE CREATED
No items to display.				

Figure 2.2: Overview of the OpenStack web interface, showing running instances and resources used [42].

2.1.3 Configuration Management

Configuration management tools within cloud computing are tools that make use of declarative resource description-based scripts to converge the system of which they run to some desired state. The name stems from the fact that such tools ensure that changes and states can be rapidly and repeatedly applied to any aspect of a system, thus ensuring that configuration happens in the manner desired by the developer [22]. Such tools are crucial parts of today's cloud environments, because it saves a lot of time compared to having to manually configure the different aspects and states for each node or update that comes to or are involved with the system. There are several prominent tools within the configuration management category, such as Ansible, Chef and Puppet, with different strengths that make them stand out. The one perhaps most familiar, however, is Puppet.

Puppet and Foreman

When it comes to the configuration management tasks in cloud environments and projects, Puppet is the preferred tool of choice for many. Puppet is a well-known configuration management tool that helps the user manage data related to configuration, meaning user information, packages used, involved processes and services, etc. Figure 2.3 shows the typical Puppet setup. The figure illustrates how the Puppet server authenticates the host nodes through SSL. Then, configurations are done through the use of declarative policies, called "manifests", where measures to ensure the functionality for the specific node is taken based on the type of the node. The control that puppet has, is made possible through the installation of an application called Puppet Agent on each involved node. Through this application, Puppet continuously evaluates the state of the node on which it is installed, before applying the manifests created at the Puppetmaster according to the needs on the node. Foreman then contributes to the setup by providing visualisations of the structure that is being applied through Puppet, as illustrated in Figure 2.4, showing various visualisations formats. The manifests, in many ways function as recipes that the agent on each node uses to investigate what changes, if any, need to be done on the node. Because of this system, it is easy to make changes quickly and have them enforced on a large quantity of nodes with minimal effort, as the agent installed on each node will make sure the changes occur, without the developer having any involvement at the individual level. However, Puppet also has ways of categorizing so that not all changes have to be applied to all nodes. This can for example be done through the use of hostnames, which let the developer or system administrator select which units specific changes will apply to [47].

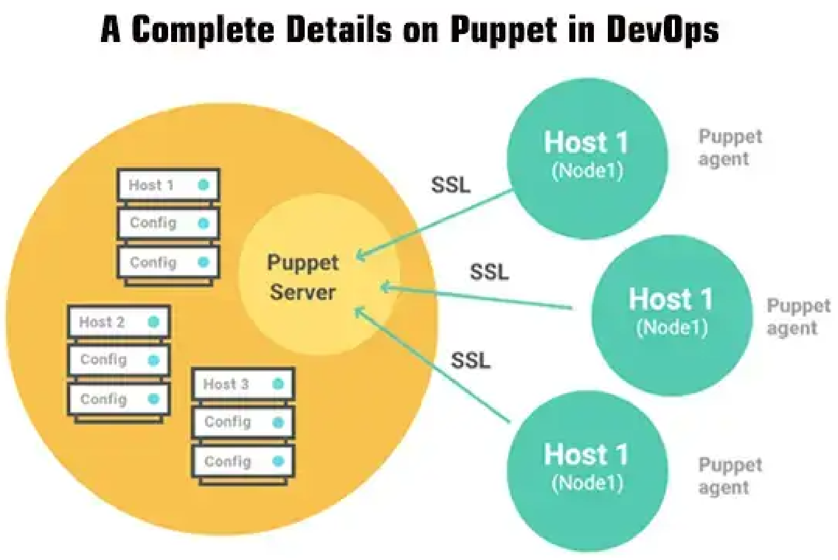
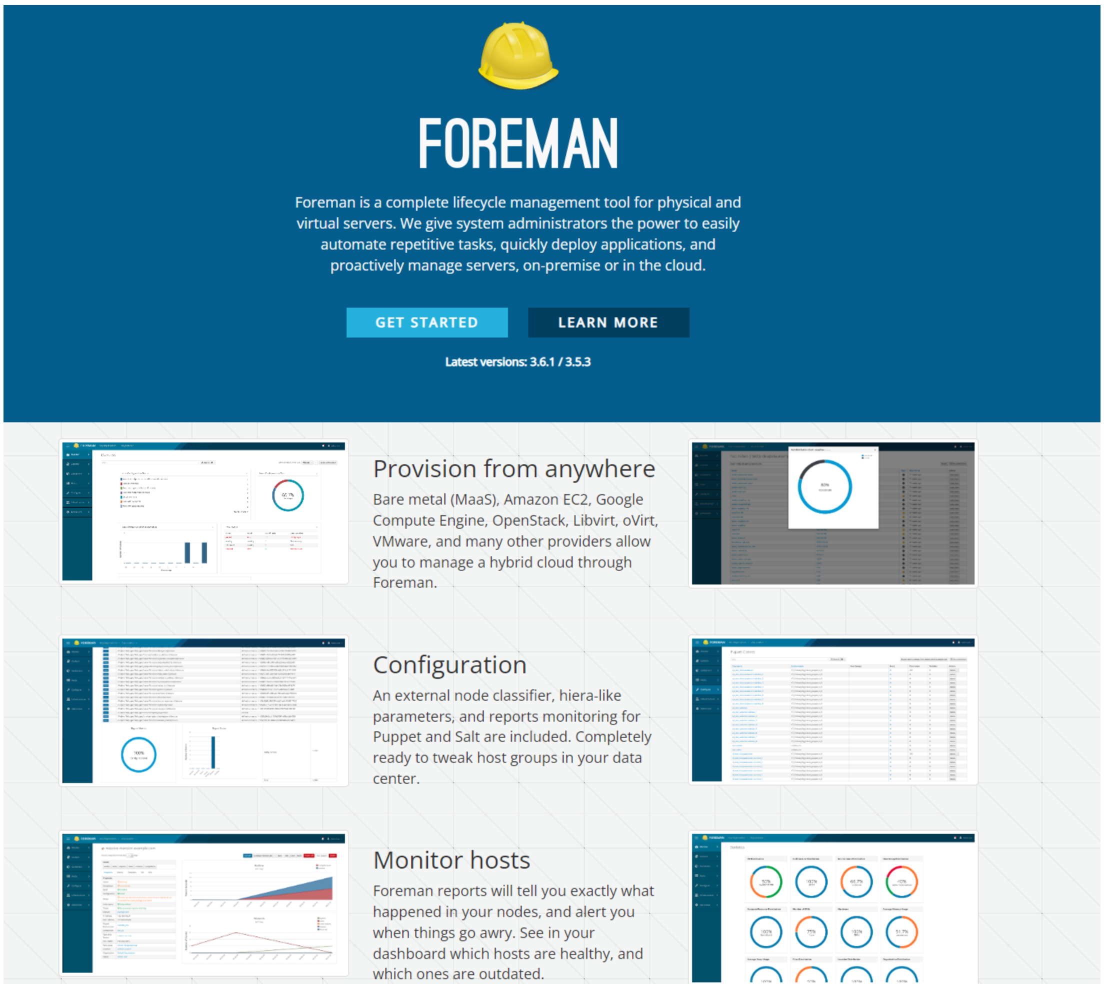


Figure 2.3: The typical structure of Puppet [33]

Many developers elect to go with Puppet, not only because it contains all the most crucial functionality that is needed for cloud projects, but also because it is a configuration management tool of which almost every person within the field has heard about and have knowledge of to some extent. This makes documentation and problem solving or information gathering easy, with tons of guides and information readily available. One could thus argue that selecting such a well-established tool is time efficient in the sense that selecting this tool often means saving time through not having to learn the functionality of a different, less well-known configuration management tool. Many also like the combination of Puppet and Foreman, a web-based interface which can be used to get a more instructive and intuitive overview of the system it entails. In this sense, Puppet feels like somewhat of a complete deal, in that it provides both the configuration management itself, while also easily being able to be combined with another tool for a holistic monitoring setup as well.


FOREMAN

Foreman is a complete lifecycle management tool for physical and virtual servers. We give system administrators the power to easily automate repetitive tasks, quickly deploy applications, and proactively manage servers, on-premise or in the cloud.

GET STARTED LEARN MORE

Latest versions: 3.6.1 / 3.5.3

Provision from anywhere

Bare metal (MaaS), Amazon EC2, Google Compute Engine, OpenStack, Libvirt, oVirt, VMware, and many other providers allow you to manage a hybrid cloud through Foreman.

Configuration

An external node classifier, hiera-like parameters, and reports monitoring for Puppet and Salt are included. Completely ready to tweak host groups in your data center.

Monitor hosts

Foreman reports will tell you exactly what happened in your nodes, and alert you when things go awry. See in your dashboard which hosts are healthy, and which ones are outdated.

Figure 2.4: The Foreman website, with various visualisations [19].

2.1.4 Virtualization

In the world of computing, virtualization can be seen as the abstraction of computer resources. It is an aspect of cloud computing which enables developers to separate the operating system of the hardware on the computer or unit that they are working on and split this into several virtual resources. Virtualization also allows for the reverse option, allowing several physical resources to make up one virtual resource. The concept of virtualization has made big improvements to cloud computing, allowing a higher degree of energy saving, decrease costs and reduce the hardware footprint, amongst other things [28]. In a solution combining cloud deployments and CI/CD pipelines, then, it would still be worth it to look into options for virtualization of the solution. Here, amongst several options, Docker stands out as the perhaps most common one.

Docker

In a large portion of the cloud environments utilizing virtualization, one finds that Docker has been selected to play a role. Usually, this has to do with the fact that Docker enables the environment to make better use of the available resources. Docker is a containerization tool released in 2013 that has risen to prominence in recent years and are now one of the most well-known tools for such tasks [11]. In short, Docker can be seen as a light version of a virtual machine, which allows a developer to build applications that can then be run anywhere using containers. The architecture can be seen in Figure 2.5, illustrating how a docker image is built using instructions from a Docker file. Images can also be uploaded to registries online, which in turn makes it possible to pull the images for later use. These images can then be used to deploy docker containers.

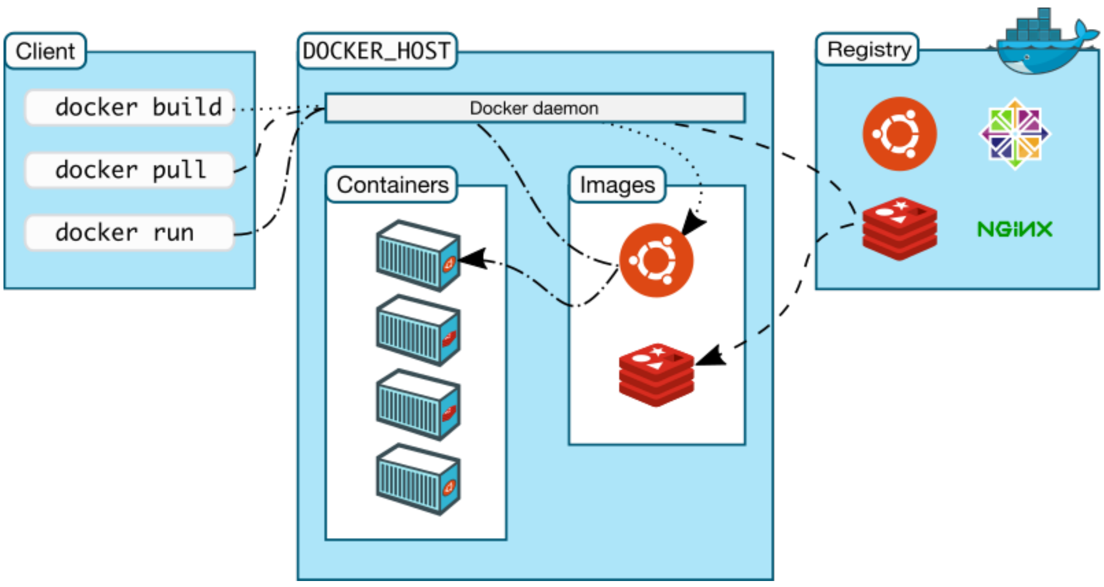


Figure 2.5: Illustration of the general Docker architecture [32]

There are several benefits to implementing docker containers as part of a cloud solution. Firstly, it is more resource efficient than running the solution "plainly" on a single virtual machine or server, in that a plethora of Docker containers can be deployed and run on the same single virtual machine, meaning that the containers use less resources than a virtual machine does. This is in turn beneficial because a virtual machine is a larger computational resource that also requires longer time and is somewhat more complex to deploy. Since many projects start off as relatively small, they are often to be run with relatively restricted resource access. Because of this, virtualization is often a very appropriate tool in making sure that the scarce resources that are available for the projects are being utilized as efficiently as possible. Running the setups on containers could also be beneficial in reducing the risk of environment dependencies, or, in other words, that the solutions end up only working within the specific environments in which they were created. This is a very common problem, particularly within the DevOps movement, as addressed by Charles Andersons [2] 2015 publication. Docker, then, is already known as a very suitable tool for use within DevOps-related operations, of which continuous integration and continuous delivery are often a part. Additionally, implementing Docker also gives the possibility to utilise some of the other features known from the technology, such as seamless container portability, automated creation of containers based on source code within the applications, container reuse through image rebuilding etc [11].

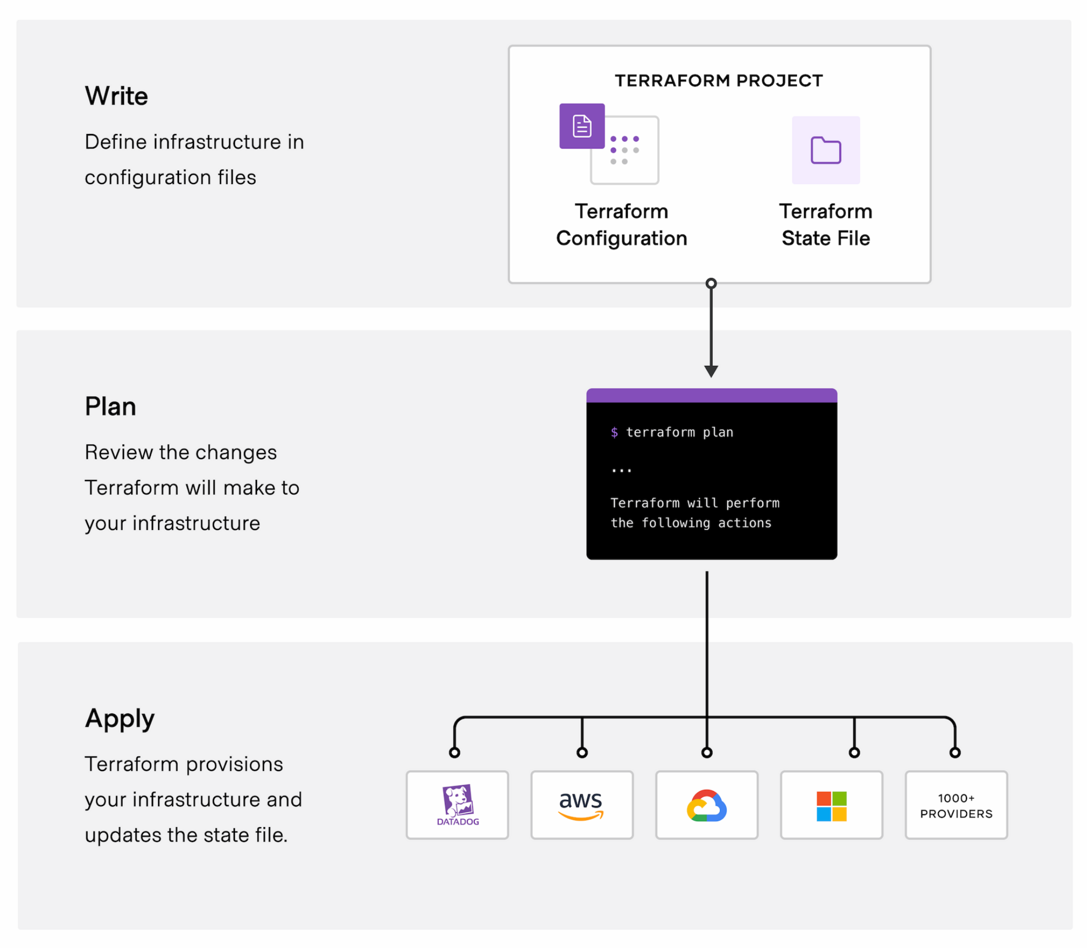
2.1.5 Orchestration tools

While the cloud technology has established itself as a major force on the market of computing, some of the problems related to it are still significant in their nature and have been almost since the creation of the field of cloud computing. Some of these issues, like vendor lock-in and lacking or downright poor tool support, can lead to it becoming hard to take full advantage of the dynamic nature and scaling capabilities of the technology, some of its most dominant features. It is because of these issues that orchestration tools, tools that target seamless management and orchestration of cloud resources, have risen in prominence [4]. These tools make resource orchestration a lot easier, and today contribute to the automation of many of the processes and steps that were previously locked behind manual labour. Amongst these tools, we find Terraform.

Terraform

Since a core concept with any cloud creation project that strives to be successful today involves some degree of automation, one of the most crucial building blocks required for cloud environment projects is to select a tool that could be suitable for such a task. For this task, a lot of companies elect to go with Terraform as their tool of choice. Terraform is an infrastructure as code tool, meaning in short that it is a tool that utilizes the process of computational management through definition files that are readable by machines. This is also the key to the automation aspect, because these definition files enable computers to handle the infrastructure management and provisioning, instead of it being handled through manual processes. Since its release by HashiCorp in 2014, it has become one of the leading tools in the market of infrastructure provisioning and management, with support for over 30 different providers, a special configuration language which lets you declare the desired infrastructure configuration in text-based templates as well as a solid, albeit somewhat complex foundation for resolving dependencies, intelligibility and reliability through a graph logic system. It is also compatible with a lot of the different configuration management systems currently available and easy to extend through plugin implementations, to mention some of the benefits [50].

At its core, Terraform works by setting up a three-step workflow as illustrated in figure 2.6, using fundamentally the same concept as regular configuration files:


Write
Define infrastructure in configuration files
TERRAFORM PROJECT
Terraform Configuration
Terraform State File
Plan
Review the changes Terraform will make to your infrastructure
Apply
Terraform provisions your infrastructure and updates the state file.
DATADOG
aws
1000+ PROVIDERS

```
$ terraform plan

...

Terraform will perform
the following actions
```

Figure 2.6: Model of Terraform workflow [24]

The write-step can be viewed as a definition phase, where resources are created and can span across different cloud infrastructures. This can for example be a configuration file that deploys a virtual machine or installs an application.

The plan-step follows the write step and involves Terraform creating a plan of execution that contains descriptions of whatever it is supposed to work on. This work can involve creating, updating or deleting the resource, depending on the instructions given in the configuration or infrastructure.

The final step in the workflow is then the apply step, where Terraform utilizes the setup from the previous planning step to perform the desired actions. This also involves committing these actions in the order they were described, thus ensuring that no potential dependencies are broken. This could for example involve rescaling of an entity, such as a virtual private cloud (VPC), if the content within this entity is changed.

One of the reasons why so many companies chose to utilize Terraform for their projects, is that it is such an established tool in today's market, with utilities that allow for many of the core features one would want from such a tool, like infrastructure management, standardization of configurations, infrastructure tracking and more. Additionally, it is known as one of the more advanced options on the market, while also having relatively low barriers of entry in terms of learning. For example, in their 2020 study, de Carvalho and de Araujo [13]concluded that Terraform was the most suitable tool to be used for Sky Computing, specifically because of its high level of maturity at present time, in combination with a higher utility range than most other options.

While the previous section has covered core concepts of cloud structures, the way they are linked up together has not yet been addressed in much detail. Thus, as an initial phase to the prototype creation, an effort to gain a higher degree of clarification on previous work on the topic and how other researchers have taken in their attempts to make similar creations was made. This was both due to the fact that the knowledge here would be valuable when setting up a prototype of our own, and because it is evident that automation in the field of cloud technology is still lacking in some respects. Because of this, the following section will start off with a clarification of previous works, with a categorization of the different approaches, their goals and outcomes.

Chapter 3

Approach

As addressed in the problem statement, the aim of this project is to explore the possibilities of using continuous integration in deployment of cloud infrastructure, with regard to cloud operations. In other words, the core idea is to investigate if a very well-established and popular concept in the field of agile development can successfully be applied to the deployment stages of another very popular and fast-growing technology, cloud computing. As briefly mentioned in the previous segment, this idea is something that has been discussed for several years within the industry already, but many parallels can be drawn between this case and the concept of self-driving cars. Much like with CI/CD in cloud deployments, self-driving cars have been an area of investigation for many years, with car companies, being led by Tesla, expressing great desires to strive towards complete autonomy for cars in the future. However, while the concept is clearly sought after within the industry, and discussions are rampant about how it can be done, no such car yet exists, and companies have reported regular issues in development on a regular basis. While the technology is closer to the futuristic concept that we've all imagined than it has ever been before, there are still glaring uncertainties and issues that make it hard to completely trust in the system of the car, with things like temporary signal failure, misconfigurations and other things making up roadblocks for the proper realization of the self-driving car. In fact, one need look no further than to The Dawn Projects testing report revealed in 2022 [43], to see that things are not all on track, with the self-driving Tesla apparently failing to recognize child-sized, stationary mannequins when going at an average speed of 25 MPH, or 40KMH.

Self-driving cars, then, illustrates how a concept that seems ideal on paper can have aspects to it that makes it less ideal in practice, or at least that it will require a lot of time and effort to reach this ideal state. However, the car industry has not given up, and lots of research and resources are still being invested into figuring out how this can be made possible in the future. In the same fashion, while a complete setup and deployment of cloud infrastructure with CI/CD pipelines seem difficult at this point in time, this is not to say that a solution does not exist out there. Thus, the core idea behind this project will be to explore the possibilities of integrating CI/CD pipelines into the deployment process of Cloud Infrastructure deployments. To go through with this idea in a successful manner, we intend to split the project into two phases.

The first phase will consist of a collection and categorization of relevant literature and examples of previous works, to see how other administrators in the industry and researchers researching in the field have made attempts at similar concepts that is being investigated here. This will then give an overview of technologies and strategies used, as well as enable comparison of different strategies, what worked and what didn't, and see this in light of the next phase of the project.

The next phase will consist of taking the knowledge learned from the observations we have made to create a prototype of sorts, building on the previous works of other researchers gathered, to study if the model could contribute to the building of future implementations of such solutions. This will involve some degree of testing to study how the prototype works compared to how it was intended and our goals.

Finally, the findings and data from these phases will be summarized in a final phase, where the findings are compared and studied. The outcome of the project and the solution to our problem statement. An inclusion of ideas and considerations on where to take our research further in the future will also be present.

3.1 Phase 1 – Categorization of previous work in the industry?

To make the process of categorizing the previous works as efficiently as possible, a table would be created with predefined categories in place. The categories were made based on the values or key aspects of the solutions that were made in the projects investigated, while also being the ones deemed most crucial for us when making our own prototype. This creation marked the beginning of the first phase of the project, and the categories were:

The goal of the project

The goal with the project, meaning the ideas and aspirations the researchers themselves had for the project in question. This section was added with the idea that knowing the background for the project could both help determine the motivation for the project, as well as uncover potential ideas aligning with our own ideas for our project.

Proposed Pipeline

The proposed pipeline or flow that has been created by the researchers, the solution itself. Here, information on the pipeline was gathered and entered as a stepwise instruction list, making it easy to get an overview of the different steps going into the complete pipeline.

The outcome of the project

The outcome or the project, and whether or not the group reached their goals. This was based largely on the conclusions and testing done by the researchers of the respective project but could also include potential observations uncovered while studying their work, to uncover and counteract potential biases.

The complexity of the project

The complexity of the project. In other words, the idea with the category was to get an overview of how easy the project was to grasp, mainly in terms of the structure of the pipeline and the quality of the report. In this case, then, this is meant as the project as well as the description of it, how clear the research is, for example. Along with the reproducibility-category, this category was created with the intention of helping produce an overview of how easy a project and its solution was to grasp, making it easier to figure out a less time-consuming starting point when creating our own prototype.

Reproducibility of the project

How reproducible and/or adaptable the proposed solution seems to be. For this category, we looked for mainly how easily one could go through the research done in the project and replicate it, and with how easy it would be to be successful in doing so. This could, for example, be how flexible the solution is to different tools, if it is dependent on a particular environment etc. This category was important in that the classification in this category would be a major contributor to the decision-making process when drawing inspiration for our own prototype, because it would help us determine and estimate of how much time and effort would have to go in to setting up a solution based on on of the entries in the categorization.

Implemented tools

The tools in play in the solution. This category was again divided into different sub-categories with the typical tools previously mentioned as crucial for a cloud environment to work appropriately:

- CI/CD
- Cloud models
- Configuration management
- Virtualization/ Containerization
- Orchestration tools

In addition to the concrete categories, a column for the ***title of the works***, as well as a column for the links to the ***resources of the articles*** or works themselves will be added, mostly for organizational purposes, making it easier to manoeuvre through and look up specific resources if something needs to be revised or more information is needed further down the line. The different categories were developed with the idea of prototype creation in mind; with an extensive description of the pipelines, combined with the technologies used for the different sections in the respective pipelines, the idea was to make it easy to discover which technologies and combinations worked well, giving a better overview of which features and projects to use for a prototype of our own. The complexity and reproducibility/adaptability sections were made as descriptions of the respective projects to map out which ones would be easiest to draw inspiration from when developing a prototype ourselves, with a colour coding system added for these categories; A green colour code would mean that the project had a relatively low level of complexity and high level of reproducibility. A yellow colour code meant that the work had a moderate level in the categories, while a red colour code would mean that we found the work to be highly complex or having a low degree of reproducibility associated with it. The setup is illustrated in Table 3.1.

Colour code system			
Complexity	Low level of complexity	Medium level of complexity	High level of complexity
Reproducibility	Easily reproducible	Moderately difficult to reproduce	Hard to reproduce

Table 3.1: Colour coding system for the categorization

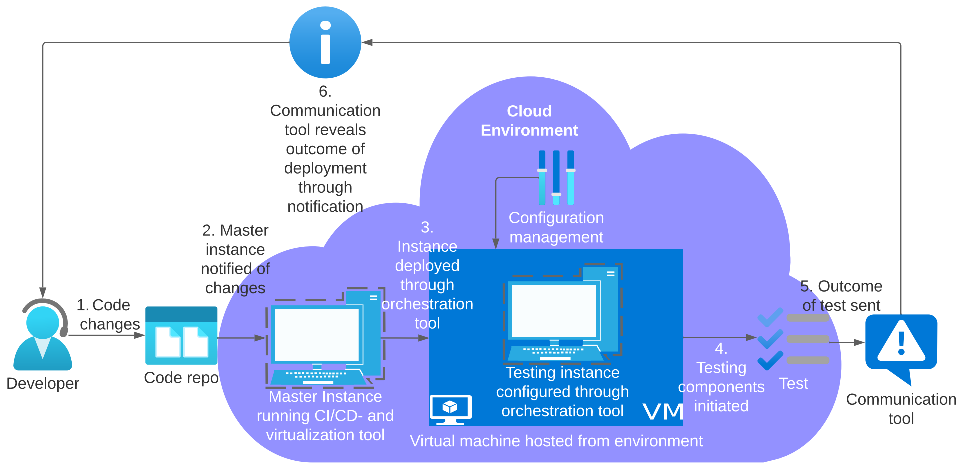
The idea was that a project having a seemingly great solution could lose a significant portion of its value to us if the project has a high level of complexity and low level of adaptability, e.g. a large reliance on a particular technology or a poorly written description of the pipeline or a large pipeline with a large portion of different technologies. By this we mean that we in our second phase of the project aimed to investigate the implementation of a CI/CD pipeline ourselves, and thus needed a good overview of available technologies that also seemingly had a high degree of adaptability. This is because we wanted to make sure that a solution were robust enough to withstand alterations to the various parts of the pipeline, as well as to make sure that inspiration drawn from previous work could actually be put to use in our own project, something that would have been made difficult with a high degree of complexity that we would need time to properly understand, or a reliance on specific technologies that we might not have the same access to in our project. These columns were also color-coded according to the conclusion made after researching the projects, making it easier to compare the outcome of the different projects. This made up the classification system to be used for the project, which is illustrated below in Table 3.2.

Categorization		
Title		Project 1
Goals		
Pipeline		
Outcome		
Complexity		
Reproducibility		
	CI/CD	
	Cloud Type	
Tools	Virtualization	
	Configuration Management	
	Orchestration Tools	
Source		

Table 3.2: Categorization of Projects Studied

3.2 Phase 2 – Basis of a Model Implementation

After conducting the research and subsequent classification of previous works, the second stage of the project will be the implementation of a general model to study how the concept could be applied. As with the self-driving car-concept highlighted in the beginning of the chapter, developing knowledge on how the current state of the research on the field is crucial to get a grasp of how the concepts can be combined in a valuable way. By investigating and categorising other works, we will obtain knowledge that can be valuable in several ways. To highlight this, an illustration can be used.



6. Communication tool reveals outcome of deployment through notification
Cloud Environment
Configuration management
2. Master instance notified of changes
3. Instance deployed through orchestration tool
1. Code changes
Developer
Code repo
Master Instance running CI/CD- and virtualization tool
Testing instance configured through orchestration tool
VM
Virtual machine hosted from environment
4. Testing components initiated
Test
5. Outcome of test sent
Communication tool

Figure 3.1: Initial draft of a CI/CD pipeline

During the initial phases of the project, different drafts were made as a thought concept of how a CI/CD pipeline implementation could potentially work, where the idea – in very general terms - is that a developer makes changes in the code, leading to a master instance in a cloud environment deploying an instance that functions as a testing environment. Once the instance is deployed, the tests are committed, and the result is notified to the developer through a communications channel. While the figure above shows an initial draft of a CI/CD pipeline set up during an early stage of the project, where we had outlined our proposed tools in the categorization but hadn't't gone through the actual categorization process. Thus, the figure shows the idea of the pipeline as believed that it could function before doing our research on previous works. Because of this, while the different stages and rough steps in the pipeline are there, no technologies are specified. This is also where the categorization process and its first form of value comes back into play.

Firstly, the categorization process will give us an overview of how similar projects and ideas have been attempted previously, and, by extension, which technologies were utilized in these projects. This will make it easier to determine which technologies were often used and which ones that were often scrapped. This is beneficial both in terms of highlighting the benefits of the most popularly used ones, and in terms of ease of obtaining documentation on the technologies selected, because it will be easier to select technologies that has been thoroughly investigated and utilized previously.

Secondly, while the figure above shows an initial proposal to a complete pipeline, without prior knowledge on how such pipelines have been attempted to be implemented before, the pipeline is still subject to changes, with a high probability that more beneficial setups have been missed. Through the categorization of previous works, such improvements and changes would also prove easier to discover, as we will develop a sound foundation on how the pipelines previously attempted has been constructed, with the reasoning behind the designs as well as potential changes or improvements the researchers would have done in the future.

Chapter 4

Results Phase 1

After setting up the categorization, the project commenced by doing exploratory work on the field to uncover works that had been handling the same subject matter in previous efforts. The emphasis was put on articles and projects that had tried to create a proposal or solution of a CI/CD pipeline, in combination with some use of a cloud related environment or technology type. The thought here was to ensure a high degree of relevancy through the selection of project aspects that directly correlated to our own. For each potentially useful project discovered, an entry was made in the categorization, before the search continued. Then, after having found a solid base of works and placed them in the categorization, each work was thoroughly visited and researched. If the work was found irrelevant or too lacking in terms of technologies used, i.e not having utilized most of the categories in our tool section, it was removed from the list. If it was found to be a relevant piece of work, each category were filled out, with the complexity and reproducibility categories being added with descriptions and color codes depending on our validation of the levels within the categories. The entries that ended up being included in the categorization, along with the results of the work, will be addressed in the following section and shows a high degree of variation both in terms of the projects themselves, the technologies used as well as the scored given in the categorization.

4.1 Categorization of Projects

The following section illustrates the various results of the project categorization phase after all the selected projects were put into the format illustrated in Figure 3.2. Each table contains detailed information about the various projects, along with our ranking of the projects in terms of complexibility and reproducibility. This ranking also displays the colour coding system explained in Table 3.1 in the approach chapter.

4.1.1 Project 1

Project 1		
Title		CI/CD Pipeline to Deploy and Maintain an OpenStack IaaS Cloud
Goals		1. Build a pipeline that worked with packaged OpenStack while giving them the ability to build a multi node development environment that people could use on their personal workstations. 2. Use their own public cloud infrastructure to build test systems for pipeline validation?
Pipeline		1. Individual Salt Formula development 2. Personal multi-node, package based OpenStack environment on workstation for dev and validation 3. Go public – push to gerrit for review and automated testing 4. Pull in to the deploy-kit 5. Deploy-kit tooling kicks in and builds deploy artefacts 6. Auto deploy toi ephemeral public cloud test environment 7. Deploy to physical staging environment 8. Ready for production
Outcome		Pipeline has allowed them to build environments and systems that allows them to validate the data that goes to built other environments. Validate the configuration data of production environments
Complexity		Relatively high, pipeline contains far more steps and layers than many of the other entries in the list. Deployment split into several phases and involves deploy artifacts. Pipeline also contains manual aspects that they would like to get rid of long term. Also not great at handling repackaging (of Ubuntu packages); Applies packages directly to the Python scripts.
Reproducibility		Project itself is well documented, but does not contain a concrete step-by-step guide on how on would redeploy the pipeline. Instead covers the parts included and their function. Pipeline also relatively rigidely tied to the specific context of the project, but should be adaptable in terms of changing out or at least running more modern versions of some technologies.
Tools	CI/CD	1. Git, Gerrit, gitshelf and Jenkins 2. Test-kitchen (kitchen-salt, serverspec)
	Cloud Type	OpenStack Through Ubuntu 12.04
	Virtualization	SaltStack
	Configuration Management	Vagrant/ VirtualBox, public cloud
	Orchestration Tools	SaltStack
Source		[9]

Table 4.1: Categorization of Project 1

The first project that was included in our categorization was the 2014 project by Simon McCartney [9] and his accomplices, presented during the

2014 OpenStack Summit. The project predated the HP Helion OpenStack and utilized Ubuntu 12.04, in combination with OpenStack Grizzly and SaltStack (McCartney, 2014). The background for the project is not mentioned in detail, during McCartneys presentation, but he does mention the details known with the CI/CD pipeline technology in general, such as reapeatability, reliability, a constant flow of changes etc (McCartney, 2014), and that these benefits were something they wanted to try to take advantage of in combination with their current cloud infrastructure. Thus, the goals are mentioned as being two-fold; firstly, they wished to build a pipeline that worked with packaged OpenStack while giving them the ability to build a multi node development environment that people could use on their personal workstations. Secondly, they wanted to base the project on their own public cloud infrastructure and use this to build test systems for the validation of the pipeline. This objective was achieved, as the outcome of the project was that the pipeline created allowed the team to build environments and systems that allowed them to validate data going into the building process of other environments, as well as validate the configuration data of the environments ready for production. The table above shows the components they used in their project to make their goals achievable, as well as the categorization we made in terms of complexity and adaptability. The project has some benefits in that they utilize several technologies that are not only well-established as valid solutions in their respective fields, but also something we have previous experience with. This could be a benefit in terms of time saving, in that less time would have to be invested into understanding how the pipeline is to be set up and function. However, we rated it relatively poorly in our categorization, as evident by the table. Firstly, the pipeline contains a relatively large number of steps to function, with some aspects being manual. This is something the researchers themselves addresses and makes clear that they want to get rid of. Combined with the pipeline being poor at handling repackaging and the fact that the project as a whole is largely tied to the specific context of the working environment of the researchers, it scores relatively poor in terms of the key categories of complexity and adaptability.

4.1.2 Project 2

Project 2		
Title		Set up a CI/CD pipeline for cloud deployments with Jenkins
Goals		Goals not mentioned as the article is a guide more than a research project.
Pipeline		1. GitHub sends code through web hook integration 2. Jenkins start running automated builds and tests after each code check-in 3. Sample web application is deployed as part of CI/CD pipeline, accessable for end-users through container engine for Kubernetes cluster 4. Terraform used for infrastructure automation
Outcome		Outcome not very relevant given the context of the article.
Complexity		Relatively low, a straight forward pipeline structure with well-known terchnologies, well desrcibed.
Reproducibility		Very high reproducibility as this article is more of a guide than a project, thus with a goal of just that. Code for pipeline is even included in GitHub repo for complete reproducibility. Adaptability also seem high, as the technologies are well-known as well. Biggest hindrance might be the use of Oracle Cloud.
Tools	CI/CD	GitHub, Jenkins
	Cloud Type	Oracle Cloud Infrastructure
	Virtualization	Jenkins
	Configuration Management	Kubernetes
	Orchestration Tools	Terraform
Source		[10]

Table 4.2: Categorization of Project 2

The second entry differs from the rest of the entries in the categorization, mainly because it cannot really be considered a research project. Instead, it is a thorough documentation from Oracle [10], last changed in 2022 and focused on how to set up a CI/CD pipeline for cloud deployments, using Jenkins. While this leads to the document lacking information in some of our classification categories, such as the goals and outcome of the "project", the technologies used and concept outlined in the documentation is still highly relevant, enough so that it was determined that it had enough value to be included in the documentation. While lacking in some categories, it scored quite well in the complexity and reproducibility categories in our table. This is because it utilizes well-known technologies to make the pipeline work, technologies that are also properly documented. The pipeline itself also contains fewer steps and less complexity than the McCartney pipeline, making it easier to develop a proper understanding of. In terms of reproducibility, it draws a massive benefit from the fact that the entry is a guide more than a research project, making reproducibility the closest thing to a goal that the document has. In fact, the documentation states that the code for the pipeline can be fetched directly from a repository in GitHub, illustrating the good score given to the entry in these categories. The biggest drawback in terms of reproducibility might be the use of Oracle Cloud, as the document does not detail how the solution works or can be adapted to different environments or ran locally.

4.1.3 Project 3

Project 3		
Title		A Kubernetes CI/CD Pipeline with Asylo as a Trusted Execution Environment AbstractionFramework
Goals		1. Create a complete CI/CD pipeline that addresses some of the well-known issues with existing implementations 2. Create a solution with a focus on adaptability and reproducability
Pipeline		1. Developer initiates the CI pipeline by committing and sharing changes 2. Event listeners automate each successive action in the pipeline, from building the artifact to storing it for future use. Tekton automates each of these actions. A event listener checks out the latest commit and builds an image using a Dockerfile in the repository. Building the image is performed by Kaniko, a tool to build containers inside a Kubernetes cluster, using Asylo to ensure the confidentiality of the image. 3. Upon success, the finished artifact (i.e., the image) is pushed into Harbor, a container registry scoped locally to the Kubernetes cluster. 4. If all these actions complete successfully, the second pipeline stage begins execution immediately; otherwise, the CI pipeline fails, denotes the cause of the failure (i.e., the action that failed), and stops execution.
Outcome		Managed to create the complete pipeline with increased emphasis on security thanks to Asylo,reaching the main goal of the project. The success is a bit more up in the air as the researchers conclude that further testing is needed to know how well some of the other aspects, such as adoptability, work
Complexity		Some level of complexity in the solution having a multiple-step pipeline setup. However, well documented, with descriptions of the technologies as well as the reasoning behind picking them.
Reproducibility		Project is very well documented, with clear figures and descriptions for each part of the pipeline and the project. For the adaptability, the suggested pipeline utilizes largely well-known and well-established technologies, as well as open source projects. This has been done specifically with adaptability in mind. The researchers also describe the factors that could prove challenging when adapting to other systems, trying to ensure these steps being as seemless as possible.
Tools	CI/CD	Tekton/ asylo
	Cloud Type	Google Cloud Platform
	Virtualization	GitHub
	Configuration Management	Kubernetes
	Orchestration Tools	Not mentioned
Source		[34]

Table 4.3: Categorization of Project 3

The third article is a 2021 research project by Jamal Mahboob and Joel Coffman, describing a CI/CD pipeline created with Kubernetes and used as a trusted execution environment abstraction framework. The authors state in the article that the utilization of CI/CD pipelines vary greatly, with some glaring issues being apparent on a general basis (Mahboob and Coffman, 2021). One of the areas they find lacking is the area of security, which is why they have put extra emphasis on this topic with the creation of their pipeline, using Asylo to cover this area. Asylo is a development framework for trusted execution environments, or TEEs, which offers systems as “wrappers” around the code in question. This in turn allows developers to make use of such

TEEs without having to restructure their own code bases [34]. The main goals were, then, twofold: the researchers wanted to create a complete pipeline that contained aspects addressing the known issues occurring with current implementations. Additionally, they wanted the solution to be created with a focus on reproducibility and adaptability. The focus of the project thus distinguishes it from many of the other projects mentioned in this article, in that it is more centred around improving known issues with CI/CD pipelines, as opposed to many of the others that focus on making new pipelines based on old concepts. The project also distinguishes itself by using Google Cloud Platform as its environment of choice, being the first project utilizing this cloud vendor. As with other entries, however, Mahboob and Coffman also chose to utilize GitHub, a common occurrence for many of the projects. In terms of complexity, the project was ranked with the yellow colour code, because it has a somewhat complex, multiple-step pipeline setup that would require some time to fully grasp. The pipeline steps are however fewer than the proposed pipeline of McCartney, for example, and the setup is described in detail in the report. This documentation also plays a role in the good score given for the adaptability category, in combination with the pipeline using largely well-established technologies and open-source projects. The main concerns when adapting the pipeline to different systems are also specified and described by the researchers in the documentation, displaying their focus on adaptability for the project as a whole. To summarize, the researchers seem to have completed their project in accordance with their main goals, as they managed to create a complete pipeline with increased emphasis on security through Asylo. They do however address that some measurements for the success are uncertain at the point of their completion, and that one would need further testing to ensure the level of success completely.

4.1.4 Project 4

Project 4		
Title		Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects
Goals		1. Create a tool to improve delivery timeline in agile projects 2. Create an effective and efficient way to handle Agile-based CI and CD projects through automated pipelines, everything from generating them to customizing and automating them.
Pipeline		1. Pipeline generator 2. Versioning 3. Build 4. Deploy
Outcome		Created a solution which ensures high availability with no downtime, fast and easy scalability, automatic rollbacks and reactions based on actions and events of changes. Also contributes to higher speed on pipeline duration.
Complexity		Quite complex, several steps and very detailed descriptions for each step, sectioning the pipeline into different sections in the paper. This does however seem well documented to compensate.
Reproducibility		Project is very well documented, with clear figures and descriptions for each part of the pipeline and the project. For the adaptability, the suggested pipeline utilizes largely well-known and well-established technologies, making it more robust for different projects. One limitation might be the dependency on AWS.
Tools	CI/CD	GitLab
	Cloud Type	AWS
	Virtualization	Git
	Configuration Management	Docker ran in Kubernetes cluster
	Orchestration Tools	Helm, orchestrates the Kubernetes cluster on which the environment runs
Source		[14]

Table 4.4: Categorization of Project 4

Another project utilizing technologies like GitLab and Kubernetes is the fourth entry, where researchers Donca et al. [14] proposed a method for CI/CD using a pipeline generator for agile software projects. The background for the project was that the researchers wanted to improve the delivery timeline of agile software projects, and thus wanted to create a tool responsible for this improvement. As a second goal, they wanted to create a way of handling agile-based CI and CD projects that was effective and efficient, through automated pipelines, spanning from generation to customization and automation. The pipeline was run on Amazon Web Services (AWS), making the fourth unique cloud type in the list. As previously mentioned, however, many of the other technological components were the same or similar to previous entries. The pipeline itself was relatively complex, as it was sectioned off into different parts in the report. Despite solid documentation, this still placed the complexity level in the middle category. For adaptability, however, the score was very good. This has to do with the previously mentioned, very solid documentation of all components used, including illustrations and figures. The proposed pipeline also followed the trend of utilizing well-known technologies, increasing the robustness of the project. As with previous entries as well, the one initially striking dependency that could pose some trouble in terms of adaptability was the use of AWS, since this is a paid service. In terms of the outcome of the project, however, the researchers seem to have reached their goals, having created a solution that is ensuring high availability with zero downtime, scalability that is fast and easy, automatic rollbacks and change actions- and event-reactions. The researchers also state that the speed on the pipeline duration increased with the proposed pipeline, making it a successful project based on their criteria [14]. The fact that this project dealt with the development of a pipeline generator specifically, made the article interesting not only because of its relevance to our project, but also because it opens up some interesting potential in terms of merging these technologies with another concept that are currently experiencing a massive surge in popularity, artificial intelligence (AI). The ability to generate pipelines for a project could in time enable AI to specialize in tailoring automated workflows to the specific criteria of the project, further increasing what can be done and specialized in the development industry.

4.1.5 Project 5

Project 5		
Title		A continuous integration system for MPD Root: Deployment and setup in GitLab
Goals		Setting up a CI system within the available infrastructure of MPD Root project.
Pipeline		1. Windows Azure servers 2,1. Server 1 – Used for deployment of second server with version control system 2,2. Server 2 – CI-server 3. GitLab CI Runner – Execution of build and test tasks 4. Docker – Isolation of parallel CI tasks from each other.
Outcome		Reached their goal of creating the solution
Complexity		Very simple and straightforward solution. Few steps, well known technologies.
Reproducibility		Hard to reproduce, little information about the project and poor documentation in the report. Hard to adopt because of the project being tailored to specific root MPD project, even though technologies used are well-known.
Tools	CI/CD	GitLab CI Runner
	Cloud Type	Windows Azure servers
	Virtualization	gitlab-ci-multi-runner package on build and test server
	Configuration Management	Docker
	Orchestration Tools	GitLab Community Edition package on separate version control server
Source		[16]

Table 4.5: Categorization of Project 5

The fifth entry in the categorization describes the creation of a continuous integration system for MDP Root, utilizing GitLab for deployment and setup. The authors of the article, Fedoseev et al. [16], present their research in a similar manner to the second entry in the categorization, the oracle documentation. Their main goal of the project is to set up a CI system using the infrastructure available to them through the MDP root project, something that they mention in brief that they manage to complete. The proposed pipeline is straightforward and quite simple, resulting in the project getting a good complexity score. The documentation as a whole is however quite poor, in the sense that details about the project itself is lacking. Additionally, the project is tailored specifically to the assets of the MDP root project, meaning that the project did not score very well in terms of the reproducibility category.

4.1.6 Project 6

Project 6		
Title		COaaS: Continuous Integration and Delivery framework for HPC using Gitlab-Runner
Goals		1. Implement a CI/CD framework in the development of workflows through GitLab runners 2. To build an ecosystem for the university and their stakeholders that facilitates the transfer of knowledge, expertise, and solutions into the region and beyond. 3. Ensure that the environment is adaptable for end-users through the GitLab runners.
Pipeline		1. Development – Functional and non-functional requirements 2. Features are pushed to GitLab 3. Features are separated into different categories based on different stakeholders and thus moved to different sections within the environment, but all are sent in one single yaml-file. GitLab runners use containerization to allow for different features to be active at different times 4. Features are sent to the access layer, giving end users access
Outcome		Created a solution serving the purpose they had in mind, while also being able to containerize the solution
Complexity		Quite high complexity. Splits pipeline into different sections, involving both containerization in some sections and HPC in other parts. More complex than other solutions in the list, but also quite extensively covered in the report.
Reproducibility		Seems quite hard to reproduce. Strategy is described in detail, but not much mention about the use of some technologies that we deem crucial in our project. Also involves HPC for tasks involving algorithms, something we are not particularly focused on within this particular project. Adaptability hampered by the solution being tailored to a specific project at the school in question.
Tools	CI/CD	GitLab CI Runner
	Cloud Type	Local research lab
	Virtualization	Built-in features of the GitLab runner
	Configuration Management	Docker ran within the GitLab runners
	Orchestration Tools	Mentions the use of yaml-files,no details about the context
Source		[49]

Table 4.6: Categorization of Project 6

Sharif et al., the researchers behind the 2020 project called "COaaS: Continous Integration and Delivery framework for HPC using Gitlab Runner" and the sixth entry in the categorization distinguish themselves from the rest of the entries, in that they present a bigger and more extensive set of goals for their project. Their intention was primarily to implement a CI/CD framework within the development of workflows, through a branch of GitLab called GitLab runners. In addition to this, they wished to build an ecosystem for their university, as well as the stakeholders facilitating the transfer of knowledge, expertise and solutions into the region and beyond [49]. Lastly, given the success of these two goals, one can distinguish a third goal being that they wish to ensure that the use of GitLab runners make the environment adaptable for end-users. According to the article, the researchers not only feel like they achieve this, but that they were even able to go beyond their goals and containerize the solution through the use of Docker. While the solution created seems like a solid foundation for a potential prototype of our own, the project scores quite poor in both the complexity and reproducibility categories. The reason for this is that the pipeline has quite a high level of complexity, where the pipeline is split up into different subsections, where some parts are containerized while others contain High Performance Computing (HPC). While HPC has become very relevant to many projects these days and can be used to great effect when trying to increase testing speed, implementing it provides another layer of complexity to the project, thus making it a costly concept to implement. This could also be a determining factor in why this solution is the only one using it out of all our entries. Reproducing the environment also seems quite difficult, because of the use of HPC, as well as the use of a local research lab as their environment of choice for running the project. While a logical choice given the tailoring of the project to the specific university in question, this is still quite hampering for the adaptability of the solution. The reliance of HPC also poses a greater time investment, while not being something we deem necessary for our own prototype.

4.1.7 Project 7

Project 7		
Title		Implementation of a Continuous Integration and Deployment Pipeline for Containerized Applications in Amazon Web Services Using Jenkins, Ansible and Kubernetes
Goals		Create an automated CI/CD pipeline for deploying a Java-based web application in AWS, while trying out less established combinations of technologies in the solutions.
Pipeline		1. Changes in GitHub source code triggers CI job, which clones the repository in the local workspace 2. Local workspace code is compiled in a .war-file with Maven and sent through SSH from Jenkins to Ansible 3. Ansible executes first playbook, which pulls custom tomcat image from DockerHub, adds .war-file to specific folder, builds new image and pushes back to DockerHub. 4. Jenkins then triggers the CD job, executing two new Ansible playbooks. First one creates the Kubernetes specific deployments, while the second one creates Kubernetes services. Configuration files for the cluster written in yaml and stored in different GitHub repositories. 5. Downtime prevented through the use of rolling-update strategy.
Outcome		Managed to create a solution that corresponded to the "requirements" they set as their goal. Solution also seemed to be quite successful, as their experimental results showed that the proposed solution is reliable, having 0 seconds downtime, being easily scalable and fast.
Complexity		Quite high complexity, but very well covered any explained in detail in the report. The different steps in the pipeline are also relatively straightforward, so most of the complexity comes from the use of several different technologies. These are, however, well known technologies.
Reproducibility		Very well described pipeline in the report, both in the form of a description of the general architecture as a whole and more specificly through a thorough description of each step in the pipeline process. Biggest drawbacks in terms of reproducability is that it relies on several major technologies that requires some time to get in to. Also relies on AWS, which could present somewhat of an adaptability problem in our case.
Tools	CI/CD	Jenkins/ Ansible
	Cloud Type	AWS
	Virtualization	Jenkins
	Configuration Management	Docker ran in Kubernetes cluster
	Orchestration Tools	Ansible
Source		[7]

Table 4.7: Categorization of Project 7

The final article that was investigated was the research of Cepuc et al. [7], where they discuss the implementation of a CD/CD pipeline for containerized applications in AWS. Early on in their article, the researchers mention their desire to look at new ways to combine technologies for such a pipeline, or, in other words, create a solution by utilizing less well-established combinations for the pipeline. Thus, the main purpose of the project was to create an automated pipeline for the deployment of Java-based web applications in AWS, but to do it through less tried-out combinations of technologies. To accomplish this task, they used a combination of Jenkins and Ansible for the CI/CD part of the project, with Jenkins also serving as the component for configuration management and Ansible as the orchestration tool. The solution is thus distinguishable from the other entries in that it is the only solution that uses none of the Git-tools. It does, however, share a trait with other entries in its use of Docker and Kubernetes for the virtualization aspect of the solution. According to the researchers, they not only managed to achieve their goal with the project in terms of structure, but also had a measurable, performance-related success, as their testing of the solution revealed high reliability, with zero second downtime, easy scalability and good speed [7]. In terms of complexity, the project also scored well. This is in part because the different aspects of the pipeline, albeit numerous, were relatively straightforward. All aspects were also extensively covered in the documentation, with the biggest contributor to complexity being the use of an extensive variation of tools. These tools were, however, all tools with a good standing and a plethora of documentation in today's market, somewhat countering this issue. The reproducibility score of the project was also good, largely due to all steps of the project work and subsequent solution being very well-documented, both as a whole and for each individual step. The biggest potential hindrances to this would be the time required to get into the different technologies, as well as the use of AWS. Thus, although the reproducibility rating was good, this is something that should at least be considered.

4.1.8 Phase 1 - Summary of Results

The categorization process led to seven different projects being researched and categorized according to the previously described categories. The findings show a solid spread in some of the categories, while other categories show that the projects in large have some similarities, particularly in the tool selections they have done. The goals of the projects are very similar, in that most of them have a main objective of creating a CI/CD pipeline to be used to deploy some sort of cloud related project. The main variation here seems to be in terms of what specification this pipeline should have. Some projects emphasize creating the pipeline based on specific technologies such as OpenStack or resources connected with the MDP Root project, while others focus on the assignments of the pipeline. An example of this is the project of Cepuc et al. [7], focusing on creating a pipeline for containerized applications in Amazon Web Services.

Looking at the categorization, another interesting finding is the distribution of tools used in the different projects. For example, in terms of cloud solution selection, there is a large spread of different options being selected. Here, while the recurring theme is that the major cloud providers are the ones largely being selected, the spread between these are still prominent, in that AWS, Google Cloud Platform and Windows Azure are all selected. While these are all paid options, the free open-source OpenStack solution has also been selected for one of the projects, illustrating the large spread within this tool category. The selection here shows that all the researchers have elected to go for premade and well-established solutions as their cloud environments as opposed to making environments of their own or running the projects locally, solutions that could be beneficial in terms of for example costs, but that would require more time and effort. Another factor here could be that such pre-established environments often come with some crucial features embedded in the system. This could in turn be the reason why several of the articles are lacking in their description of the use of orchestration tools, one of the categories we set up as crucial in our categorization.

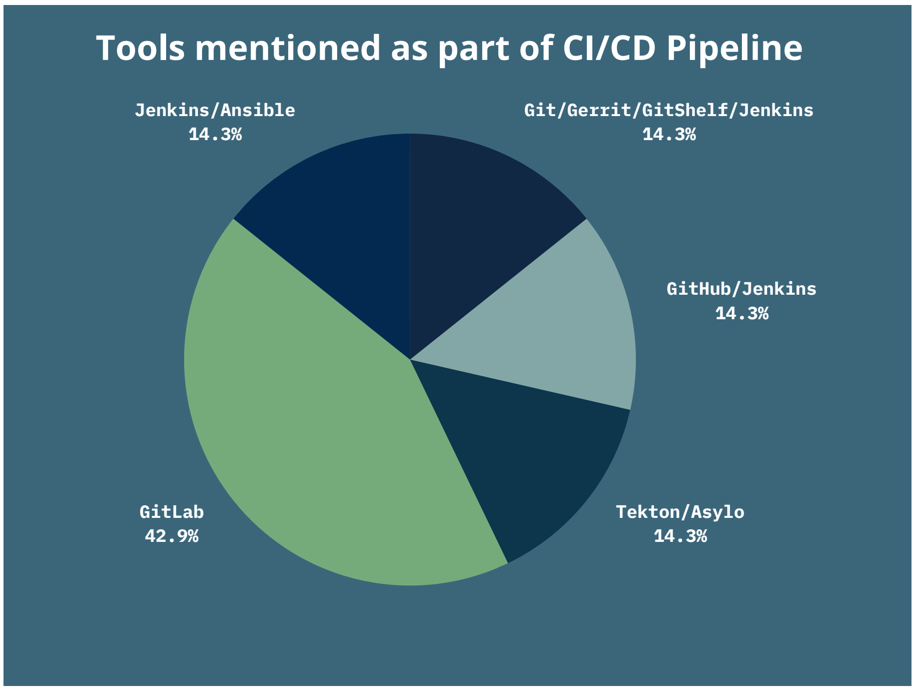


Figure 4.1: CI/CD tools used in previous projects

On the other hand, in the CI/CD category, this spread is far less varied, with almost half of the projects electing to go for GitLab as their tool of choice, and another project selecting Git as theirs. This trend is clearly visible in the graph in Figure 4.1. This goes to show how leading GitLab, and its built-in CI functionality has become on the field today. The most striking project in terms of this tool category is the research of Mahboob and Coffman [34], electing to go for Tekton and Asylo as their CI/CD tools, technologies that are less well-known in comparison to GitLab. This does however make sense to some degree, given how their project has a particular focus on addressing known issues that other well-known CI/CD implementations have. They do for example have a particular focus on securing their solution, which is where Asylo comes into play, making their selection of tools, albeit more uncommon, quite reasonable for the particular project.

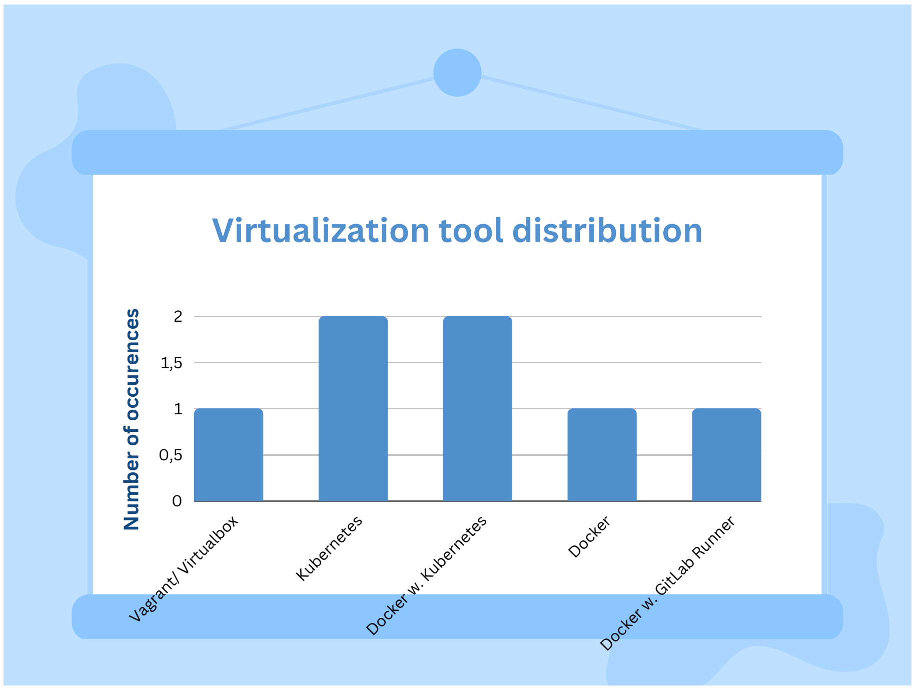


Figure 4.2: Virtualization tools used in previous projects

The tool category that has the most similarity across all projects is the virtualization tool category, with five out of six projects utilizing Docker or Kubernetes as their tool of choice for the job, visualized in Figure 4.2 above. This seems logical, given how popular Docker has become as a virtualization tool in today's landscape. The only project not utilizing one or both of these technologies are the project of McCartney [9], instead electing to go for Vagrant/ VirtualBox as their virtualization tool of choice. This could be due to this project being drastically older than most of the other entries in the categorization, with the project being developed in 2014. This is very close in time to the release dates of Docker and Kubernetes, with the technologies being released in 2013 and 2014 respectively, and thus not being anywhere near as well-established as tools on the market as they are today. Another potential reason is that the McCartney implementation was not a test but a real-time project that were already in place, around which they tried to build a CI/CD pipeline. Thus, this project was a good example of how field implementations of a cloud cannot be entirely virtualised, and that some tools and applications require physical infrastructure. Because of that, those need to be taken into consideration when implementing automation into a system. Because of this, even though the conference presentation was relatively old at the time of our research, we felt it was relevant as a good, solid example of operations in the industry.

When looking at the outcomes of the projects as a whole, there is a clear trend that makes up cause for optimism for the project. By this, we refer to the fact that all projects investigated share the trait of accomplishing their main goals with each work. While the motives for creating the CI/CD pipelines varies for the different works categorized, one of the main goals for each project is none the less to ensure successful creation of the pipeline, something all seven entries report doing. This is also the case for the works that was categorized as having a high level of complexity and/or low level of reproducibility, with reports of the pipelines here having benefits like being able to ensure zero downtime, being successfully containerized and providing validation for other systems, amongst other benefits [14][49][9]. These results of higher speeds go hand in hand the benefits generally associated with the implementation of CI/CD pipelines, and conclusions drawn in other projects, such as the works of Singh [53] where development speed was increased by implementing CI/CD in the development process. The categorization, then, illustrates that even works entailing aspects of high difficulty holds a clear value in terms of research, and can be worked on further to improve on the concept in the future. Additionally, the variance in terms of tools selected in the different projects highlights proves that these types of projects can be solved in multiple ways and with multiple tools. Thus, a good takeaway is the idiom that all ways lead to Rome; there are several ways and options to create and implement such pipelines, with all of them providing value in some respect.

After categorizing everything, several articles seemed promising in terms of drawing inspiration, with the green color code for at least one of the fields of adaptability or complexity. The green in our case meant that the research was found to be promising in terms of ease of understanding and potential for adaptability, usually because of solid documentation and/or the use of broad, well known and documented technologies within the pipeline. With the categorization out of the way, the work then moved to the creation of a prototype of our own.

Chapter 5

Results Phase 2

As the categorisation process was completed, the attention was shifted to the next part of the project, the prototype creation. As previously addressed, some drafts and ideas were put to paper at an earlier stage, as shown in Figure 3.1. Now, these were revisited. However, they underwent considerable re-considerations and changes, as they were now revisited with the knowledge acquired through the categorisation process in mind. The results derived from our work in the previous phase can be divided into two main categories. The first one addresses potential structural revisions possible for the pipeline, detailing what we found to be the most beneficial structure for the project, given the knowledge gained from studying previous works. This will be elaborated on in the below sections. The other category, and the perhaps most influential one here, is the tool category, where the component decisions were made. With our initial theoretical draft of the pipeline, the scope was quite general, containing information on which components we deemed necessary for the pipeline and where these could possibly be introduced, but no information on which tools would be used for these different tasks. After going through the categorization, however, the foundation was laid for a proper decision to be made on which tools we deemed most appropriate for the task.

5.1 Addressing potential pipeline structures

Prototype creation

Before completing the tool selection process, we drew up the implementation design of the pipeline itself. After doing some iterations on the initial plan, we eventually landed on a design that worked in accordance with our initially drawn prototype, illustrated in the Figure 3.1. For the final design that ended up being used for the prototype, the components and steps are shown in Figure 5.1 below.

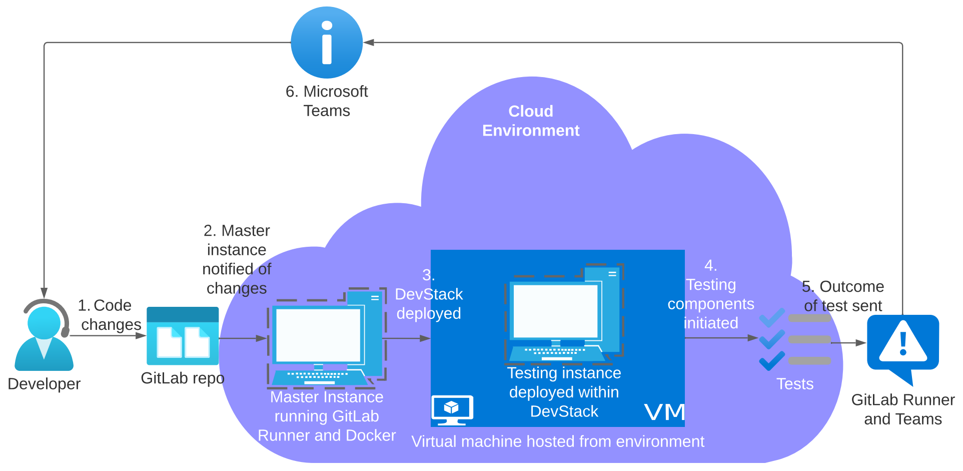

6. Microsoft Teams
Cloud Environment
2. Master instance notified of changes
1. Code changes
Developer
GitLab repo
3. DevStack deployed
Master Instance running GitLab Runner and Docker
Testing instance deployed within DevStack
VM
Virtual machine hosted from environment
4. Testing components initiated
5. Outcome of test sent
Tests
GitLab Runner and Teams

Figure 5.1: Final version of the CI/CD pipeline

Setting it up this way ended with the pipeline being divided into five different sections in GitLab, as shown in the illustration in Figure 5.2 below.

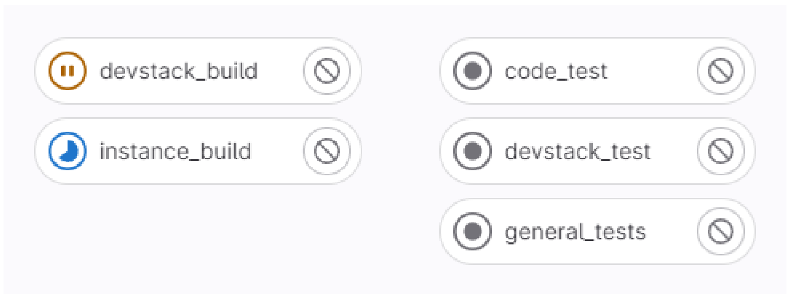


Figure 5.2: The pipeline setup in GitLab

While the final pipeline implementation ended up quite similar to our initially proposed design, but now with clear distinctions in terms of technologies and placements. One big difference in the design approach, however, is that it was decided that the emphasis on configuration management and orchestration tools. For this project, the idea was to create a small, working prototype to illustrate one way of how a cloud infrastructure deployment could be automated, meaning that the idea was to keep the prototype relatively simple, with little complexity. After researching previous works in the industry, it became evident that while doing an implementation with all our proposed tools would cover a wider range of tasks, this would come at the cost of far higher deployment time and higher complexity. Through test deployments, we for example found that adding a Puppet/Foreman setup to the implementation, alone would increase the deployment time with around 20 minutes.

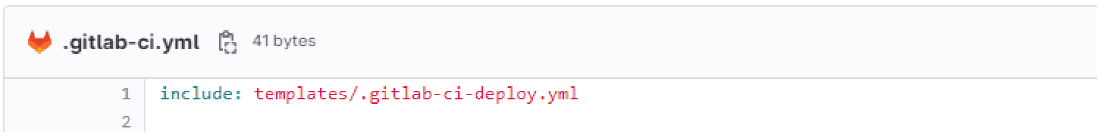
Secondly, while Puppet and Terraform have great functionality for their respective areas, another somewhat problematic aspect with their implementation was that these areas were beyond the scope of the time frame of our project, once again given the nature of the prototype. Since the idea was to deploy a cloud infrastructure and check that this infrastructure is working as intended, one could argue that the main use areas of these tools come more into play if one were to start deploying applications and system to this deployed infrastructure. This is highlighted by Puppets own documentation, declaring that Puppet is used to describe the desired states of the systems in the infrastructure [26]. Thus, while the core idea behind adding these tools remain the same, the decision was to put their implementation on hold for now, something that will be further discussed in the discussion section.

The biggest outlier to the final version of the pipeline being implemented is the use of DevStack instead of deploying invdividual instances. We decided to go for DevStack as an illustratory example, because it does a good job of showing how the environment can be deployed. According to OpenStacks own documentation, DevStack is used interactively as a development environment and is these days very common to use for much of the functional testing of various OpenStack projects, making it a very good fit for our prototype case as well [41]. Utilizing DevStack also opens up various possibilities to run tests on the environment for future endeavors, the reason for this being that DevStack upon successful deployment installs and sets up the following components:

- Keystone
- Glance
- Nova
- Placement
- Cinder
- Neutron
- Horizon

The components being highlighted above are the crucial components in making the deployed environment work. Because of the plethora of components being set up with the installment, one has the chance of running tests to check the availability of the components that constitutes DevStack individually.

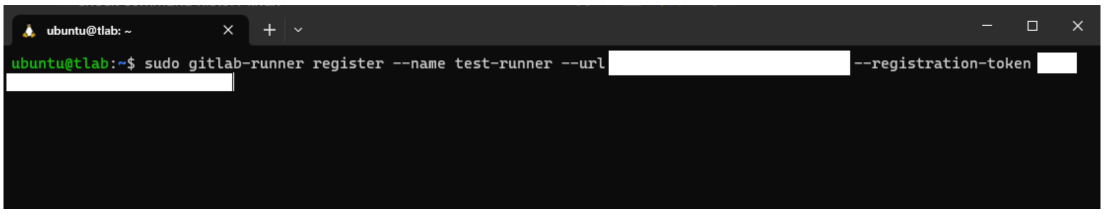
The pipeline is quite straightforward; A repository on GitLab is created and connected to the virtual machine (VM) used, by adding the machines SSH key to the repository. From there, the main YAML-file is created at the root of the repository, in this case in the main-branch.



```yaml
include: templates/.gitlab-ci-deploy.yml
```

Figure 5.3: GitLab main file setup

As shown in the illustration in Figure 5.3, the file system is set up so that the main file only consists of an inclusion of another file, where this other file is the one containing the actual scripts to be run. This makes it easier to test different files without having to alter the entire code each iteration. A GitLab runner is registered to the repository through the VM, using the GitLab-Runner register command, including the URL and the registration token belonging to the repository.



```
ubuntu@tlab:~$ sudo gitlab-runner register --name test-runner --url [redacted] --registration-token [redacted]
```

Figure 5.4: Command to register the GitLab Runner

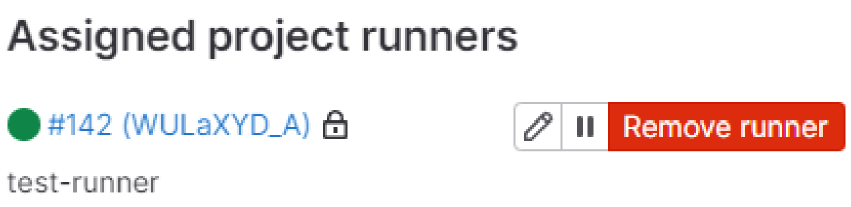
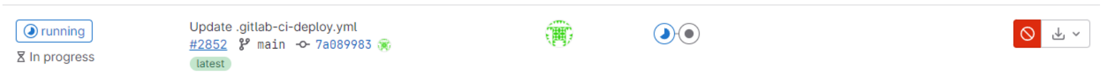


Figure 5.5: The registered runner on GitLab

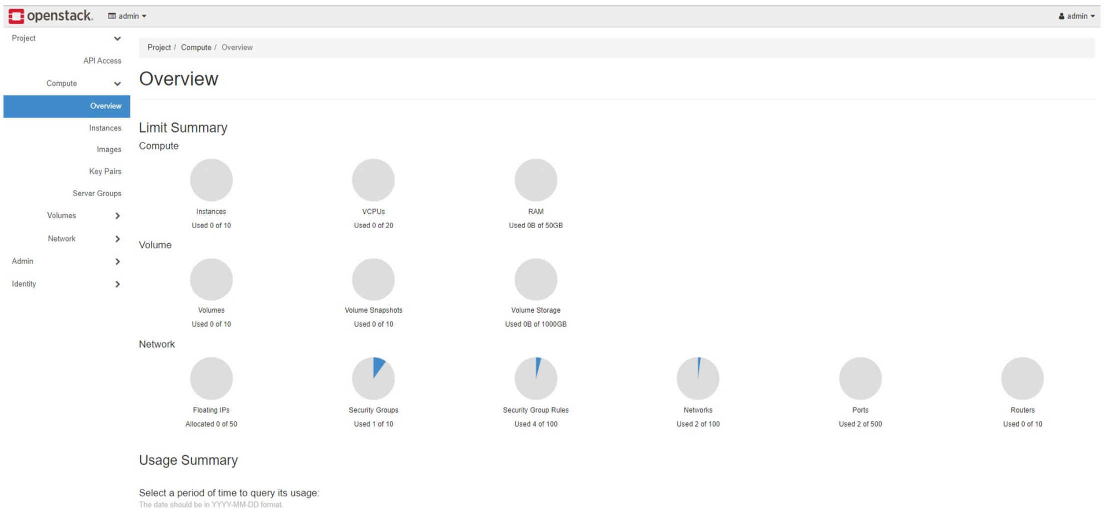
With this setup, the runner will run through the assigned script every time a change is committed to the file included in the main YAML-file. This makes it easy not only to frequently update the code and check that everything works as intended, but also to see the process in as they occur, because the GitLab interface lets you monitor everything in real time.


running
In progress

Update .gitlab-ci-deploy.yml
#2852 main 7a089983
latest

Figure 5.6: A change in the repository has activated the runner

Once the runner is set up, a script is added that has the build, test and deploy process set up. The script deploys a large instance within the OpenStack environment of the VM used as the master. This large instance is then deployed, and a simple test is run to see that the deployment has worked as intended. This test consists of running the "openstack server list" command to check that the instance appears in the list as an existing server. Once the test is completed, the script ensures that the newly created instance is logged into with SSH, before it downloads a specific version of DevStack, an OpenStack extension script collection that provides a complete OpenStack environment when ran. After the download is completed, the runner enters and starts the stack, which after around 25 minutes (manually setting up the stack resulted in a 24 minute and 23 second deployment time) setups up a fully functional OpenStack environment with a web interface, accessible through an IP address. This environment can then be used to further deploy other instances, applications etc.



Overview

Limit Summary

Compute

Instances
Used 0 of 10

VCPUs
Used 0 of 20

RAM
Used 0B of 50GB

Volume

Volumes
Used 0 of 10

Volume Snapshots
Used 0 of 10

Volume Storage
Used 0B of 1000GB

Network

Floating IPs
Allocated 0 of 50

Security Groups
Used 1 of 10

Security Group Rules
Used 4 of 100

Networks
Used 2 of 100

Ports
Used 2 of 500

Routers
Used 0 of 10

Usage Summary

Select a period of time to query its usage:
The date should be in YYYY-MM-DD format.

Figure 5.7: Overview of the established OpenStack environment

Once the environment has finished deploying, the script runs some new tests to check that the environment works as intended. For the current state of the prototype, three additional tests were created to test the success of the environment deployment. The first test investigates whether the environment is reachable by checking if the horizon page, the web interface of the environment, is reachable. This is done through the script issuing ping requests towards the internal IP address of the stack, where successful returns show that the horizon page is up and running. To get this test to work, one first has to get the output of the message stating that the environment deployment has been completed, which can be done through a GitLab environment variable, since all updates occurring with the pipeline can be read in real-time there. Fetching this message and getting the IP address from there, checking the availability of the Horizon page is a straightforward and effective test, as the ping request is a one-line command that does exactly what we need for the test.

The second test commits a deployment of an instance into the newly established environment, to check that it is fine to take into use in addition to being up and running. The script runs the "openstack server create" command, much in the same way as it did with the instance running the DevStack, and ensures that the new instance is created in the DevStack environment. Thus, the command is run from the machine hosting the DevStack, since this has the necessary authentication for access to the environment. This is crucial, because trying to deploy instances automatically with the OpenStack command will fail without this authentication. The success of this deployment is confirmed in a similar fashion to how the first test was committed, by checking the completion message and fetch the assigned IP address from this message, which can then be returned in the console.

Finally, the third test lists the available networks in the environment, using the "openstack network list"-command. For the command to work as intended, however, the Runner first ensures that the environment is logged into. Thus, the runner utilizes the previously deployed instance in the environment, by first logging into the machine using the master-VMs ssh credentials. Afterwards, like with the initial test, the command is straightforward, and the only thing needed to get the necessary network information to get the available network names. After the third task completes, everything has been tested according to desire, and the stack is uninstalled from the instance. Below is an illustratory dummy code setup, showing a rough "sketch" of how the YAML-file functions.

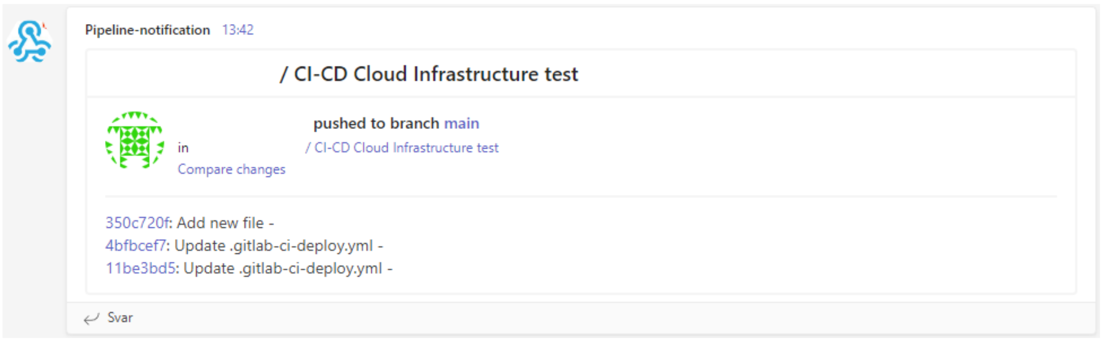
```
stages:
  - build
  - test
  - deploy

instance_build:
    stage: build
    script:
        - openstack server create -- credentials for the instance
 ↪  running DevStack

code_test:
    stage:  test
    script:
        - openstack server list -- checks that the instance have
 ↪  successfully deployed by returning all available instances
devstack_build:
    stage: build
    script:
        - ssh ubuntu@name -- enters the created instance
        - git clone
 ↪  https://opendev.org/openstack/devstack/src/branch/stable/zed
 ↪  -- Downloads the DevStack
        - cd devstack
        - ./stack.sh - Deploys DevStack
devstack_test:
    stage: test
    script:
        - echo "Stack deployed" - Basic return message to be sent
 ↪  after the stack has deployed
general_tests:
    stage: test
    script:
        - Check GitLab environment file for completion message to
 ↪  get stack IP
        - ping stack IP -- succesful means the Horizon page is up
        - openstack server create -- credentials for instance to
 ↪  be deployed on the DevStack
```

```
        - Check GitLab environment file for message to get
  ↪  instance IP
        - ssh ubuntu@name -- Log onto new instance
        - openstack network list -- checks available networks for
  ↪  network names
devstack_remove:
    stage: build
    script:
        - ssh ubuntu@name -- enters the DevStack instance
        - cd devstack
        - ssh ubuntu@name -- enters the created instance
        - ./unstack.sh -- removes the stack
```

As the final step in the pipeline, one the runner completes the task it is assigned, a notification is sent to a communications channel, notifying the developer of the changes to the system. While there are several alternatives that can be used for this step, Microsoft Teams was selected for this particular project. Setting up such a webhook requires no code and can be done directly within Teams and GitLab. For the notifications, there are several options on how this can be set up, depending on the needs of the developers. One can, for instance, only have notifications on for when an update occurs for the pipeline, specific error messages, push notifications etc. An illustration of the setup is shown in the figure 5.8 below.


Pipeline-notification 13:42

/ CI-CD Cloud Infrastructure test

in pushed to branch main
Compare changes / CI-CD Cloud Infrastructure test

350c720f: Add new file -
4bfbcef7: Update .gitlab-ci-deploy.yml -
11be3bd5: Update .gitlab-ci-deploy.yml -

Figure 5.8: Pipeline notifications sent to Microsoft Teams

5.2 Addressing potential tools

Final tool selection

After constructing the pipeline design itself, the second aspect was to select the appropriate tools to be fit into the design. To start this phase off, the focus was laid on the selection of the environment in which the creation would take place. During the initial planning of the proposed solution, the idea of running the experiment locally on a physical computer was the first option considered, as this was something we saw being done previously. The benefits of such an approach were mainly that dedicating a physical computer to the job would guarantee that the solution would be fully operational throughout the project, barring threats or accidents of some form to the computer itself, which was deemed far less likely for the project. However, this idea was discarded, as utilizing a cloud option would open up more possibilities in terms of how one could work during the project. Additionally, based on our categorization, it was clear that the decision of running the proposed pipelines on various cloud environments was far more prevalent historically. In addition to this, while all the projects studied in the categorisation reported successful creations, the project of Sharif et al. [49] utilizing the most local environment of the alternatives, ended up with one of the lower ratings in terms of adaptability and complexity, where one of the crucial contributing factors to this was deemed to be the use of such a specific environment.

The option to utilize AWS was also taken into consideration and discussed, as this was the most frequently used option in the works that were studied. Amazon is one of the leading cloud service providers in the field today, and are reported to provide great flexibility, high availability and a high degree of user control. However, it has also been reported that some of the less favourable aspects of utilizing this solution are a big learning curve, as well as all the options making it more time-consuming to figure out how to make things work. Additionally, and perhaps the main reason for us eventually deciding that the option was not the most suitable for this particular project, was that AWS was reported to have more difficult deployment and maintenance processes than other cloud services [6]. As our intention was to create a relatively small-scale prototype, this seemed unfavourable in this particular case, as time was of the essence and the ideal would be to have as much ease of deployment as possible. In the end, the choice of solution instead fell on OpenStack, because it is a well-established solution on the market, making documentation available in large quantities. It was also beneficial to make use of the free open software contained with the option, as opposed to using potentially even more well-established solutions like for example the aforementioned AWS, which provide a lot of their services for payment. We also knew that OpenStack had been used in similar projects with success before, as illustrated by the McCartney [9]project. In our case, a local solution on the OpenStack platform, the ALTO cloud, was selected, as this solution was already available from some of our previous work and was a solution that had already been tried and experienced. In other words, while we deemed a fully local solution to be less favourable, we considered this a worthwhile trade for a solution that presented previous familiarity, thus potentially saving time.

For the CI/CD pipeline, the main choices were drawn from our categorization work, but the final decision was also made based on personal experience. Looking at the categorization, the two most prevalent options seemed to be Jenkins and GitLab, as these were the tools that lead the way in terms of frequency for the previous projects. This corresponds well with the general trend in the industry, where both Jenkins and GitLab are very popular tools, bringing different benefits for both options. In their work with comparing the different CI/CD tools for integrated cloud platforms, Singh et al. [52]for example states that Jenkins is easy to use and learn, much because of its support for plugin adding. This feature makes it easy for the relevant developers to configure their respective projects for automated deployments. However, the plugin support was also mentioned as a potential cause of difficulty, because increased sizes of projects would inevitably lead to the use of more plugins, thus making management potentially more difficult.

This problem does not arise with GitLab, as creating a pipeline through this tool would see all configuration done from a single YAML-file. Because of this difference, GitLab has been argued as a better option than Jenkins in terms of freedom, because GitLab provides a simpler, but more flexible way to configure the pipeline [56]. Ultimately, however, much of the reasoning for the best option of the two will depend on the specifics of the project at hand [52]. In this case, the decision was in the end made to go for GitLab, specifically the GitLab Runner, the reason for this being mainly that the tools were something we had previous experience with to a bigger degree than Jenkins. Additionally, it has been argued that a general rule of thumb is that Jenkins is more suitable for bigger projects, while GitLab can provide a better alternative for smaller and more sophisticated projects [52].

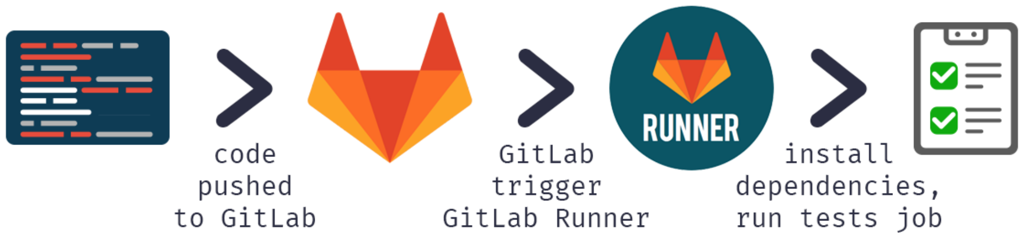

code pushed to GitLab
GitLab trigger GitLab Runner
RUNNER
install dependencies, run tests job

Figure 5.9: GitLab Runner in practice [5]

Figure 5.9 shows the typical workflow of a GitLab Runner. Because this prototype is intended to be quite small for the time being, and because of the experience previously acquired through the use of GitLab, this was then the deciding factor for the choice. GitLab Runners are very manageable to set up and deploy, as well as easy to activate so that they form a connection between the repository and the desired virtual environment, making it a great fit for the project. Figure 5.10 shows GitLabs automation process on the pipeline. This is where the illustration shown in figure 5.9 comes into play.

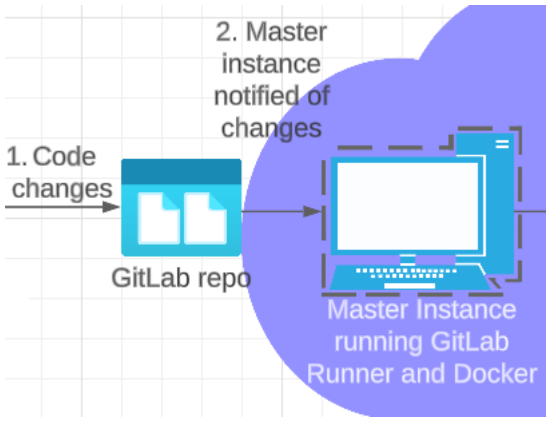

2. Master instance notified of changes
1. Code changes
GitLab repo
Master Instance running GitLab Runner and Docker

Figure 5.10: The GitLab section of the pipeline

After studying the categorization and discovering the overwhelming use of containerization in the majority of the projects, we decided that an ideal prototype would benefit from having this aspect included as well. Looking at the categorized works, it was clear that Docker was the by far most dominant option in terms of frequency, with the vast majority of projects swearing to this tool. While some of the projects used Docker in combination with Kubernetes, it was in this case determined that the use of Docker by itself would be sufficient, as Kubernetes is often used as a container orchestration tool in cases where many containers are involved [30]. As previously mentioned, since this prototype was intended to be quite small-scale, we deemed it unnecessary to deploy enough containers to make Kubernetes beneficial at this point in time. While it was initially discussed to run the setup directly on a virtual machine in the selected deployment environment, and that on could, thus, argue that the implementation was more for future benefit than strict need, a clear advantage with the implementation that could be directly beneficial for the prototype was the optimization of resource usage that follows from the containerization allowing multiple containers to be hosted on a single server [29]. Since this project was to be run with relatively restricted resource access, this aspect of containerization could therefore potentially hugely beneficial. Additionally, GitLab Runners can be initialized using predefined Docker images, another potentially great benefit. Figure 5.11 shows where containerization aspects of the pipeline, as well as the involvement of DevStack. In a real scenario, however, we would instead be deploying an actual cloud infrastructure instead of the DevStack, via the CI/CD pipeline. To do this, we would be using Terraform and Puppet as the tools for provisioning, orchestration and monitoring. Thus, instead of the virtual machine component in the figure, we would be deploying a complete cloud environment with all of its necessary components.

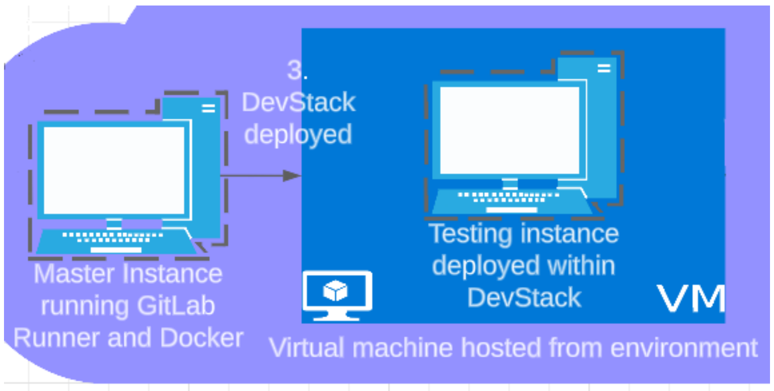

3. DevStack deployed
Master Instance running GitLab Runner and Docker
Testing instance deployed within DevStack
VM
Virtual machine hosted from environment

Figure 5.11: Docker and DevStack being utilized in the pipeline

For the selection of configuration management tools, the choice was made more based on personal experience than the categorization. For most of the works that was investigated, it was clear that configuration management was not part of the more crucial aspects of the project setups. For many of the projects, configuration management as a concept was in fact barely mentioned at all, with mostly scarce mentions of what tools served the purpose in the respective projects. Several projects also had no designated tool for the job, instead opting to utilize pre-existing features of the CI/CD tool of choice, such as built-in features of GitLab Runner. The outlier was the 2014 McCartney project where the use of SaltStack for configuration is mentioned explicitly, but as with the other projects, no big mentions of how this comes into fruition are present in the documentation. Because of this, and because of experience from previous work of our own, the selection instead fell on Puppet, a configuration management tool not used by the projects in the categorization. Even though this was the case, puppet is still recognized as a powerful tool for configuration purposes, with clear benefits like flexibility with role assignments of nodes, adding further roles for capability extension of the modules being run and ease of maintenance of the system [39]. Additionally, while deemed a suitable tool for the job, the decision to go for Puppet was also

in a sense resource based, as selecting this tool meant saving time through not having to learn the functionality of a different and new configuration management tool. The idea to set Puppet up with the web interface of Foreman for a better overview was similarly made partly because this felt like an easier way to get a good overview of the states of the nodes and the system as a whole, but also because this had been done in previous projects, so that the configuration of this setup were easily available for the project. After taking our knowledge gained from the previous works into consideration and trying to create drafts on how we wanted the pipeline to be structured, we did however decide that adding a configuration management tool to the current pipeline would require more time to investigate every component than covered by the time frame of the project, thus leading to the implementation of it being dropped. This knowledge could also be one of the reasons why the category was so seemingly lacking at most of the previous works as well, being mentioned to very little detail as previously described. There could also be other reasons for this, however, such as the projects simply not having a need for it in their creations.
Lastly, since a core idea behind this project was to look into the possibilities of automating cloud deployment themselves through APIs, we deemed it beneficial to the project to select an orchestration tool that could be suitable to contribute to the ease of such a task. For this aspect of the experiment, Terraform was selected. In addition to Terraform being a leading tool on the market because of its flexibility and adaptability [50][13], research has also been done on strategies for implementing CI/CD pipelines with Terraform as the main component. In his 2021 [36] study, for example, Modi introduced a pipeline with tasks specifically connected to tasks and steps related to Terraforms ability to build and deploy environments. In the categorization, the orchestrational tool category was then one that received the least attention from the works selection, with some entries not mentioning the use of such a tool whatsoever in their projects. The project reporting to have implemented Terraform, however, scored quite well both in terms of complexity and in terms of adaptability, which further strengthened the decision. Like with the configuration management tool category, however, like with configuration management, it was eventually decided that implementing Terraform would add to much strain to our time frame, something that will be further addressed in the following chapters.

5.2.1 Phase 2 - Summary of Results

With this setup, then, we have managed to create a prototype that demonstrates how a cloud infrastructure automation process could be executed; We set up an initial draft of a system that deploys an instance running a cloud environment upon changes to the code repository and tests that this environment is operational through deployments and tests ran towards it. The pipeline we have created is able to set up a complete and operational environment through a completely automated process, with tests included that contributes to controlling that everything has been executed and set up according to instructions. Thus, the pipeline follows the DevOps approach and consists of tools that fit into various areas of the life cycle, as evident by Figure 5.12.

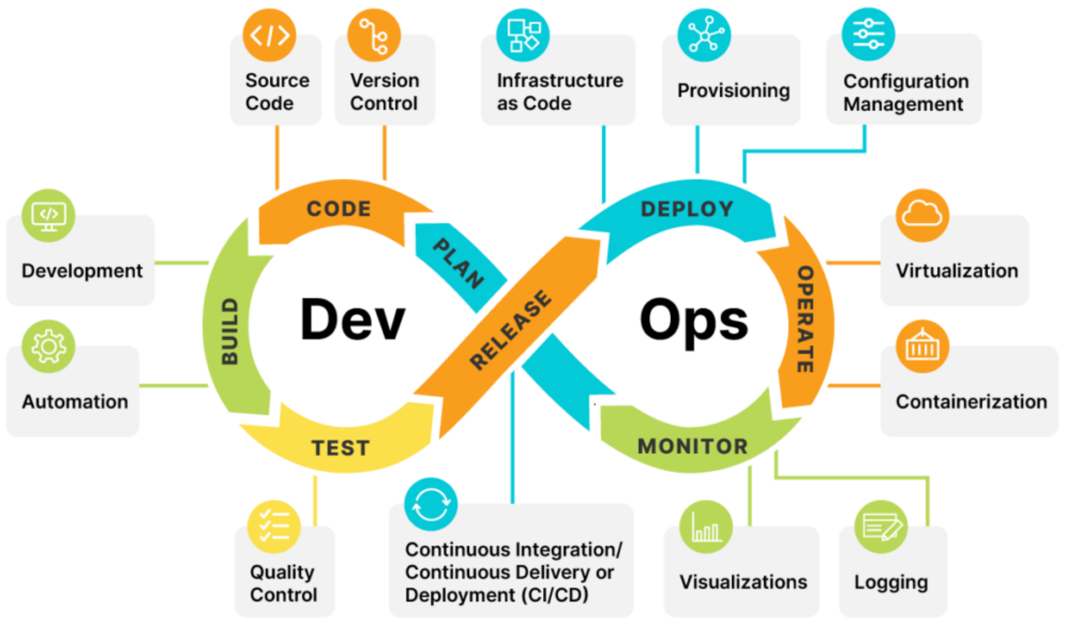


Figure 5.12: DevOps life cycle and position of pipeline tools [8].

The figure shows the DevOps life cycle split into different stages, where the various tools fit into various stages. In our first phase of the project, our research during the categorization mainly covered the build, test, plan, operate, deploy and release stages, with the different tools with had an interest in mainly covering these aspects of the life cycle. In particular, the operate-stage with Kubernetes and Docker, as well as the plan-stage with the CI/CD pipelines turned out to be crucial, as many of the previous works we investigated heavily emphasized these stages with their tool selection.

For the second phase, since our prototype was based largely on information gathered through phase 1, we have covered many of the same stages in the life cycle. We did however also address the code stage to a bigger degree than many of the previous works, going more into detail on the setup in the code repository. Additionally, with our discussions on Terraform and Puppet with Foreman, we also addressed the configuration management aspect of the deploy stage to a bigger degree, as well as monitoring and testing.

Thus, the core goal that we wanted to reach was successfully accomplished, although some aspects of our initial draft had to be altered along the way. At its current stage, the pipeline functions so that it will ignore potential dependencies between the build-, test- and deploy-stages and go through each remaining stage of the pipeline, regardless of occurrences in the previous stages. In other words, if one stage of the pipeline for some reason were to fail, that stage would be skipped, returning an error message. The Runner would, however, not stop processing the scrip there, but instead attempt to run the remainder of the stages as if nothing had happened. While this works, it is nonetheless a somewhat crude solution, and an implementation of conditioning to the pipeline to preventing from running dependent stages if a prerequired step fails would be in order. Also, while quite simple in its current stage, the prototype setup leaves options for a plethora of different and more advanced tests. This can also be linked to the aspects of our initial pipeline design that due to various reasons did not work out as intended. Thus, these are areas of improvement that we would have liked to look more into had we a larger time frame, something that along with other factors will be further addressed in the discussion section.

Chapter 6

Analysis

With the completion of the prototype, the second phase of the project was completed, marking the end of our work. Looking back at the project, we find that there are lots of things to take away from the work we have done, both in terms of the scientific results and in terms of the experiences and considerations we made along the way. While we have found that our two-phase strategy worked very well for the way we wanted to structure our work and what we wanted to achieve, one of the main takeaways from working in this manner is none the less that it comes at the cost of being demanding, as well as potentially challenging in terms of time management at certain times. As mentioned in the subsequent discussion section, one of the obstacles that we faced during our first phase of the project was that the material in terms of previous work in the industry was somewhat more lacking than initially anticipated. This made working with the classification relatively challenging, and also more time consuming than initially planned. During this stage of the project it was discussed to tackle this challenge by searching for contacts within the industry to provide us with their insights directly, something that could have further deepened our insights on the topic. Had the project spanned over a longer time period, this is an approach we would have attempted to go for, but it was ultimately scrapped in this case. Additionally, in addition to the material being more scarce and thus harder to locate, the material we did find to be of value to us were also largely of a different format than we had originally planned and hoped for. While initially designing the classification system for phase one, we wanted to find more industry based examples from conferences with access to conference videos and GitLab-repositories with access to the people behind the projects for questioning, but this proved more challenging than we anticipated as some projects we found were old, with the persons behind them no longer workng there or us not being able to gain proper access to both the idea and the project. Thus, videos matching our criteria here proved very difficult to find, with the only one providing valuable information being the McCartney 2014 OpenStack video [9]. While somewhat old in the world of cloud computing, this video also proved valuable because it highlighted that the strive towards CI/CD pipeline implementations for cloud deployments is something that has been going on for quite some time.

Because this turn of event led to the classification largely being centred around the investigation of research articles instead, our experience was that the difficulty of work increased, seeing as this presentation of the work felt more complex at the median level this way, and thus required more time to properly classify. That being said, the shift in focus that we had to make upon this realization led to us discovering new works that otherwise would have gone unnoticed. It also provided knowledge on how to classify scientific works in a way that will be brought onward in future endeavours.

Another interesting aspect with the classification work was that going into it, our expectation was that the tools used for the pipelines would be described in more detail than they turned out to be in the previous works. For several of the projects studied, some tools were only mentioned as being part of the pipelines, but no details were given on the placement of these and how they functioned in a holistic sense. This was challenging because it led to us having to draw up where we thought the tools could make the most sense to use in the context of our particular project. This selection followed the ARC-principle, stating that one can deduce that a tool has value in a particular context if one can complete an ARC statement going counter clockwise from the tool to the values it holds. The typical ARC statement is that a *tool is the way we do functions because principle for the sake of value* [23]. Thus, a determining factor when deciding on what tools to implement was to ensure that this principle could be maintained for the tools that were selected. It was also during this part of the analysis that it became clear that some of the tools that we had selected to be part of our prototype would not be very sustainable to include at this current point in time. This was largely due to some of the works being difficult to fully map out on a detailed level, while also realizing the element of too many cooks spoiling the broth; a fully developed cloud system would be of such a size and complexity that all the tools addressed in this paper provide value to keep it operational at all times. This does however come at the cost of increased complexity of automating the system, and for a prototype as small as the one we were concerned with, this cost seemed to overwrite the benefits of implementing every tool.

Because of these aspects, working with the classification phase proved to be more challenging than initially anticipated. However, the challenges also brought some rewards in the sense that each obstacle overcome led to driving the work forward in a sense, with the ultimate reward eventually being that the classification gave a very solid foundation for the second phase of the project once properly completed. Also, the concept of having to make adaptions to our work in real-time, albeit challenging, also gave some valuable insight into how to keep a constructive mindset in times where one is working uphill, in the metaphorical sense. Thus, while proving challenging at times to work with, the experience of working in such a manner was valuable both within and outside of the particular project, and also led to the classification itself eventually providing valuable insights into what tools and strategies that had previously been used.

For the technical procedures related to the prototype creation itself, going through the experiment proved to be quite a different challenge compared to previous cloud related project work. Going into the development, the idea was to set up everything with VMs located on our local OpenStack environment, develop the pipeline YAML-file and set everything up in one step through the GitLab Runner, but because of the severe instabilities of the cloud, it quickly became evident that this would not work according to plan. Theis obstacle lead to the generation and deployment of VM, of which we depended on in our initial plan, being impossible, something that lead to the prototype being somewhat more troublesome to get up to a complete state. Instead, we had to rely on a solution of checking individual components to investigate how these components, when put together, would make the whole, functional prototype. Thus, instead of our initial plan, we opted to use DevStack deployed previously on an already established VM. We went with DevStack as an example because it is a well-developed and solid package that can be used to showcase a full cloud deployment on a single VM, something that we deemed a good fit for the prototype, particularly given the circumstances. This in turn led to some other aspects becoming more difficult, through things like the need of extra authentication of the environment, the connection between GitLab and DevStack, the ability to deploy instances inside the virtual DevStack environment requiring logging into the environment through an extra step etc., and cost quite a lot of extra time both to develop each component in the pipeline and to verify that the components would in fact make up an entire pipeline. While troublesome, it did not stop our progression, however, and like with the categorization phase we gained valuable experience and insight into working with problem solving and maintaining order in a time-sensitive period, something that can never be overvalued.

Another technical challenge faced was related to the YAML file itself, this simply being that it took time and effort to understand exactly how the file would have to be written for the runner to be able to go through all the necessary steps. While this was something we expected to occur to some degree, it none the less became a time-consuming process, where repeated iterations of the file were run and studied to detect what worked and what did not, trying to fix the potential faults, before repeating the process. In hindsight, this could perhaps have been solved in different ways, although came to the conclusion that it still got the job done in an appropriate fashion, given the circumstances we were in.

Working in this manner presented some of the potential answers as to why fully automated cloud infrastructure deployments are not as of yet more common, but through the project and the eventual landing on an operational solution, it was still demonstrated that it can work. The cloud instability problem was, however, definitely an unexpected element of disturbance that we hope to not encounter in future cloud projects of similar fashion. Regardless of this, the overall experience of working with the project was positive, both in terms of the theoretical knowledge acquired from working within the particular field, and in terms of the more practical knowledge in terms of having to manage time, working on resolving obstacles along the way etc. Additionally, reaching the goals of the project also served as a confirmation that steps can be taken to push the progression of the concept further, also contributing to the overall positive experience.

Chapter 7

Discussion

While the project has led to some interesting insights, it did not go without a hitch, with several obstacles having to be overcome along the way. The first major obstacles came during the prototype creation phase, with the instance creation aspect. As the prototype was being developed, the ALTO cloud, the local OpenStack cloud environment of which we intended to base the creation, suffered from severe instabilities. These were in fact so severe, that it was impossible to create instances whatsoever. As our prototype setup required the creation of not only one, but two such instances, this proved to be a massive detriment to the progress of the experiment, as there was large difficulty trying to work around this issue with the intended structure. As this happened early on during the development phase, the obstacle was initially handled by utilizing workaround options for as large a portion of the prototype as possible. For the initial testing phase, for example, an older VM from a previous project was utilized instead of generating new ones, with some of the required components being installed directly onto this machine. In other words, the initial development phase was handled as a more split and separated process, with individual components being tested separately to a larger degree than initially intended. Options like moving to a different cloud provider like AWS or running the solution locally was also considered as plans to have as backup if the issue had not resolved itself in time. The reason for the instabilities were never fully disclosed, so it is hard to determine exactly what was the root cause and thus also hard to determine what could have been done to prevent it. In a broad sense, one potential takeaway would be to rely on more standardized solutions like several of our classified projects did, with a solution like AWS for the environment likely providing a "safer" environment in terms of stability, albeit at the cost of more resource-strain, both in terms of cost of the environment itself and in terms of time spent to get some general knowledge on how to operate the framework. Thus, some of the responsibility will have to be taken by us, as we made the decision to rely on this rather local solution, even while knowing that this could make up a potential weakness with the solution. While never anywhere near this extent, instabilities had also occurred within the environment during previous projects, something that, in retrospect, should have made up more grounds in the question of which environment tool to select for the job.

Another aspect with the project that caused some problems was what can be deemed an over-reliance on the use of tools covering every aspect of a cloud environment, based on the size of the prototype itself. By this, we refer to the fact that the prototype was initially designed with the intent of utilizing tools in five different categories, which – although without a doubt very crucial and beneficial for proper cloud environments – proved inefficient given the rather simple nature of this prototype. Since the goal of the experiment with the prototype at this point in time was to look into ways that cloud infrastructure could potentially be automated through CI/CD pipelines, there was never much emphasis to be put on the cloud environment itself. Because of this, some of the tools that would be useful, if not even required for a proper, operational cloud environment running applications or other systems within them, proved more of a source of complexity and thus, a reliability in this particular experiment. This was particularly the case with the orchestration tool in Terraform and the configuration management tool in Puppet (in combination with Foreman). The reason for this over-emphasis on covering such a wide range of use areas stemmed from our own experience working with cloud related projects before, where the importance of exactly such tools have been evident during the development work and thus something that was almost reflexively brought into this project as well. The most straightforward way to prevent such an event from happening is likely to do the same as we did in this project; to study previous work on the field and develop a better understanding of what has worked and what hasn't. Also, we could have spent more time during the early brainstorming sessions to properly go through each tool related to the prototype and properly discuss the use area of each tool in the context of the particular project, something that we will bring with us for future endeavours. In any case, the solution to this obstacle eventually became exclude these tools from the prototype, as they provided little value to it at the current iteration.

For the classification phase of the project, the closest thing to an obstacle was that we to some degree overestimated the amount of available research related to the topic of our own project. Since our prototype relied heavily on data gathered from our classification, we were depending on being able to collect ample information to make the experiment work properly, and the amount available being somewhat smaller than anticipated caused the classification phase of the project to take longer than expected. That being said, the amount of available information was still of a sufficient level to make the experiment work appropriately, meaning that while it posed somewhat of a hindrance, it is still not accurate to call it an obstacle that had to be overcome.

Overall, the project fazed relatively few problems, with the only one being a real challenge being the instabilities of the OpenStack environment. I think the overall goals were reached, with minimal delays and no obstacles devastating enough to affect the impact on the problem domain, something that will be further addressed in the conclusion section. Much of the cause for the minimal delays is credited to the approach taken for the project, which worked very good in this particular case; Although the estimation of available information was off to some degree, the overall impression is nonetheless that the strategy selected led to a solid foundation of information that could be used to determine the optimal path for the experimental phase. Completing this phase in a good way ended up saving time as compared to just trying to establish a prototype more from scratch, while simultaneously providing interesting insights into the general landscape of the field that we have ventured into with this project.

7.1 Future Work

On the topic of future work, we believe the project can be further enhanced in both phases. The categorization could be developed by including more works, alternatively one could strive to expand the criteria for the works being added to the classification, so that the scope increases that way. This would help further establish insight into the research being done in the field. Additionally, we have several ideas as to how the prototype aspect could be further developed. Since the prototype created here is functional as intended, it still has a lot of room for improvement, particularly after the decision to remove some of the tools from the creation process. The first interesting work that could be done is thus to develop the prototype into a more extensive tool, or, in other words, ensure that it deploys a more complete and robust cloud infrastructure. This could for example be to ensure that the established cloud infrastructure that currently gets deployed is supplemented with instances added to it, thus opening up for the creation of developer environments with applications readily available. Developing it into such an operational environment could in turn lead to more advanced tests, for example through checking how the infrastructure handles automatic deployment and maintenance of applications being deployed inside of it. This would also open up the possibility to have different YAML-files for different types of infrastructures being deployed, increasing the flexibility of the prototype as a testing tool, as one could change what scripts to run depending on what kind of environment one wishes to change and test.

As the complexity of the prototype increases, as well as to reiterate on our previous idea, another area that could be worked on to improve the prototype would be to look into ways of implementing orchestration tools like Terraform and configuration management tools like Puppet. While deemed unnecessary at this point in time, increasing the complexity and contents of the prototype would mean that the need for such systems would also increase. Looking at Figure 5.12 again, adding Puppet and Foreman would increase the coverage of the pipeline with regards to the deploy- and monitor-stages in the DevOps life cycle, as Puppet contributes to the configuration management aspect of the pipeline, while Foreman can be seen as a Visualization tool of sorts. More components being added would mean that automatic orchestration through Terraform would ease the implementation of such components, and the idea of adding different servers or instances to the deployed infrastructure would make Puppet useful to monitor that these servers are continuously operational. Such implementations would also be interesting to study how the implementations affect the deployments committed in the scripts. Implementing Terraform would also strengthen the build stage of the pipeline, given its ability to automate and build components connected to the pipeline.

Finally, while Docker is currently involved with the GitLab Runner of the current iteration of the prototype, another field of further development would be to study ways of further implementing containerization, through Docker to a bigger degree. This could build into the other potential improvements mentioned, for example by containerizing the applications deployed to the cloud infrastructure. This could further improve the resource usage of the system, adding to the validity of the solution. As the solution grows, one could also consider looking into orchestrating the docker containers created through Kubernetes, something that would ease the process of operating the solution when it reaches a more considerable size.

Chapter 8

Conclusion

Ultimately, the project turned out to be a challenging, but enriching experience. While working with cloud related projects was something we had previous experience with, translating this into automated cloud infrastructure deployments was uncharted territory for us. Through the initial pipeline drafts, we developed an idea of how we thought a pipeline of this fashion –at its core – could work, something that helped us get a better overview of how to proceed with the work. Through our investigation in phase one of the project, we have developed a detailed and well-structured classification of how previous projects of a similar nature within the industry has been conducted, a classification that significantly affected the latter stage of the project and provided further building blocks to move forward with our pipeline design. Through the classification work, as well as the obstacles that had to be overcome along with it, we have gained valuable insight into the different pipeline approaches that currently exists within the industry. As an extension of this, we have also explored what tools currently dominate the industry when it comes to CI/CD pipeline creations, some of which has been explained thoroughly in this paper. Additionally, this way of conducting research has itself given us valuable experience in how to go about investigating work and information on a specific topic, experience that can span far beyond the borders of this project.

After finishing the categorization phase of the project and acquiring important knowledge from this, we used this knowledge to create an example prototype, not only illustrating what tools are beneficial for such projects, but also showing how a successful implementation itself could look like. While some instabilities occurred, this did not prevent us from reaching the objective we had going into the phase, as we successfully set up a prototype of a full pipeline, and how this could work and be enhanced in the future. Being able to draw it up and illustrate how a potential solution could be made for a successful end to the second phase of the project, as well as a good confirmation that the considerations made during and because of the categorization work were properly founded in the research on the field.

In conclusion, through this project we have managed to create a proposed pipeline for automatic cloud infrastructure deployments, with implemented concepts of testing to ensure that the environment is successfully operational at the end of the deployment stage. The time frame in which we could create the prototype was very short, but because of the steps leading up to the prototype creation was completed in an adequate manner, it was still big enough to create a functional example solution. Thus, our prototype has illustrated that elements of automation can certainly be used to contribute to less human errors and more effective deployments of such infrastructures. Through the prototype, then, along with our suggestions for feature work, such as expansions on the contents added to the established cloud environment, one could argue that our work has had a positive impact on our problem domain, in that we gained knowledge in the area that we set out to explore. In the beginning of this paper, a parallel was drawn between our topic and the concept of self driving cars. While the industry related to self driving cars have come a longer way and have already began testing various implementations and releasing products, the pipeline implementations for cloud deployments are none the less something that we expect to only grow more common in future years. And it is our belief that our work can contribute to bringing new insights into new ideas and concepts on the topic, thus contributing to driving the industry further towards the ultimate goal of reaching the cloud equivalent of the self-driving car.

Bibliography

[1] Inc. Amazon Web Services. *What is DevOps?* 2023. URL: https://aws.amazon.com/devops/what-is-devops/.

[2] Charles Anderson. 'Docker [Software engineering]'. In: *IEEE Software* 32.3 (2015), pp. 102–c3. DOI: 10.1109/MS.2015.62.

[3] Mohammed Shamsul Arefeen and Michael Schiller. 'Continuous Integration Using Gitlab'. In: *Undergraduate Research in Natural and Clinical Science and Technology Journal* 3 (2019), pp. 1–6.

[4] Daniel Baur et al. 'Cloud orchestration features: Are tools fit for purpose?' In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2015, pp. 95–101.

[5] Bestdevops. *How to install Gitlab runner on windows OS*. Online; Picture taken from bestdevops.com on May 14, 2023. 2021. URL: https://www.bestdevops.com/how-to-install-gitlab-runner-on-windows-os/.

[6] K Bhargavi and Omkar Sharma. 'AWS vs. MWA: A review'. In: *International Journal of Engineering Research & Technology (IJERT)* 1.9 (2012), pp. 1–7.

[7] Artur Cepuc et al. 'Implementation of a Continuous Integration and Deployment Pipeline for Containerized Applications in Amazon Web Services Using Jenkins, Ansible and Kubernetes'. In: *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. 2020, pp. 1–6. DOI: 10.1109/RoEduNet51892.2020.9324857.

[8] Tyler Charboneau. *What Is DevOps?* Online; Picture taken from orangematter.solarwinds.com on May 12, 2023. 2022. URL: https://orangematter.solarwinds.com/2022/03/21/what-is-devops/.

[9] *CI/CD Pipeline to Deploy and Maintain an OpenStack IaaS Cloud*. 2014. URL: https://www.openstack.org/videos/summits/paris-2014/ci-cd-pipeline-to-deploy-and-maintain-an-openstack-iaas-cloud (visited on 05/11/2014).

[10] Oracle Corporation. *Set up a CI/CD pipeline for cloud deployments with Jenkins*. 2022. URL: https://docs.oracle.com/en/solutions/cicd-pipeline/index.html#GUID-D5231DA5-98CB-4690-B15F-656181B0080C.

[11] The International Business Machines Corporation. *What is Docker?* 2023. URL: https://www.ibm.com/topics/docker.

[12] Creatly. *CI/CD Pipeline Example*. Online; Picture downloaded from Creatly on May 12, 2023. 2023. URL: https://creately.com/diagram/example/c5JMedWsSpq/ci/cd-pipeline-example.

[13] Leonardo Rebouças De Carvalho and Aleteia Patricia Favacho de Araujo. 'Performance comparison of terraform and cloudify as multicloud orchestrators'. In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, pp. 380–389.

[14] Ionut-Catalin Donca et al. 'Method for continuous integration and deployment using a pipeline generator for agile software projects'. In: *Sensors* 22.12 (2022), p. 4637.

[15] Arvind For Edureka. *DevOps Life cycle: Everything You Need To Know About DevOps Life cycle Phases*. Online; Picture taken from Edureka on May 12, 2023. 2023. URL: https://www.edureka.co/blog/devops-lifecycle/.

[16] George Fedoseev et al. 'A continuous integration system for MPD Root: Deployment and setup in GitLab'. In: vol. 1787. July 2016, pp. 525–529.

[17] Pawel Flajszer. *Benefits of Cloud Deployment with Continuous Integration*. 2022. URL: https://fullduck.dev/benefits-of-cloud-deployment-with-continuous-integration/.

[18] Wallpaper Flare. *Gitlab*. Online; accessed April 26, 2023. 2023. URL: https://www.wallpaperflare.com/search?wallpaper=gitlab.

[19] Foreman. *Illustratory image of Foreman and how statistics can look*. Online; Picture taken from the Foreman front page on April 26, 2023. 2023. URL: https://theforeman.org/.

[20] Al Geist and Daniel A Reed. 'A survey of high-performance computing scaling challenges'. In: *The International Journal of High Performance Computing Applications* 31.1 (2017), pp. 104–113.

[21] Taras Gleb and Taras Gleb. 'Build Automated Pipeline'. In: *Systematic Cloud Migration: A Hands-On Guide to Architecture, Design, and Technical Implementation* (2021), pp. 161–179.

[22] Oliver Hanappi, Waldemar Hummer and Schahram Dustdar. 'Asserting reliable convergence for configuration management scripts'. In: *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*. 2016, pp. 328–343.

[23] Syeda Taiyeba Haroon. 'Investigating A Rule Re-ordering Algorithm in order to optimize Software Testing Pipelines'. MA thesis. 2018.

[24] HashiCorp. *What is Terraform?* Online; accessed April 26, 2023. 2023. URL: https://developer.hashicorp.com/terraform/intro.

[25] Lorin Hochstein and Rene Moser. *Ansible: Up and Running: Automating configuration management and deployment the easy way.* " O'Reilly Media, Inc.", 2017.

[26] Puppet Inc. *Introduction to Puppet.* 2023. URL: https://www.puppet.com/docs/puppet/6/puppet_overview.html.

[27] Ramtin Jabbari et al. 'What is DevOps? A systematic mapping study on definitions and practices'. In: *Proceedings of the Scientific Workshop Proceedings of XP2016.* 2016, pp. 1–11.

[28] Nancy Jain and Sakshi Choudhary. 'Overview of virtualization in cloud computing'. In: *2016 Symposium on Colossal Data Analysis and Networking (CDAN).* IEEE. 2016, pp. 1–4.

[29] Hui Kang, Michael Le and Shu Tao. 'Container and Microservice Driven Design for Cloud Infrastructure DevOps'. In: *2016 IEEE International Conference on Cloud Engineering (IC2E).* 2016, pp. 202–211. DOI: 10.1109/IC2E.2016.26.

[30] *Kubernetes vs Docker : A comprehensive comparison.* 2023. URL: https://www.civo.com/blog/kubernetes-vs-docker-a-comprehensive-comparison (visited on 13/04/2023).

[31] Rakesh Kumar et al. 'Open source solution for cloud computing platform using OpenStack'. In: *International Journal of Computer Science and Mobile Computing* 3.5 (2014), pp. 89–98.

[32] Open Clip Art Library. *Illustration of the Docker architecture.* Online; Picture taken from Openclipart on April 26, 2023. 2023. URL: https://openclipart.org/detail/226503/dockerarchitecture.

[33] SimpliAxis LLC. *Illustration of the Puppet structure.* Online; Picture taken from simpliaxis on May 2nd, 2023. 2023. URL: https://www.simpliaxis.com/storage/images/article_detail_banner_image_A-Complete-Details-on-Puppet-in-DevOps_1642829855.webp.

[34] Jamal Mahboob and Joel Coffman. 'A kubernetes ci/cd pipeline with asylo as a trusted execution environment abstraction framework'. In: *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC).* IEEE. 2021, pp. 0529–0535.

[35] Midjourney. Online; Picture created with the assistance of Midjourney on April 29, 2023. 2023. URL: https://midjourney.com/.

[36] Ritesh Modi and Ritesh Modi. 'CI/CD with Terraform'. In: *Deep-Dive Terraform on Azure: Automated Delivery and Deployment of Azure Solutions* (2021), pp. 163–190.

[37] Sikender Mohsienuddin Mohammad. 'Continuous integration and automation'. In: *International Journal of Creative Research Thoughts (IJCRT), ISSN* (2016), pp. 2320–2882.

[38] Adam O'grady. *GitLab Quick Start Guide: Migrate to GitLab for all your repository management solutions*. Packt Publishing Ltd, 2018.

[39] Henrik Öhman et al. 'Using Puppet to contextualize computing resources for ATLAS analysis on Google Compute Engine'. In: *Journal of Physics: Conference Series*. Vol. 513. 3. IOP Publishing. 2014, p. 032073.

[40] Opensource. *What is OpenStack?* 2023. URL: https://opensource.com/resources/what-is-openstack.

[41] OpenStack. *DevStack*. 2023. URL: https://docs.openstack.org/devstack/latest/.

[42] OpenStack. *OpenStack overview image*. Online; Picture taken from the OpenStack web interface on April 26, 2023. 2023. URL: https://cloud.cs.oslomet.no/horizon/project/.

[43] The Dawn Project. *The Dawn Project's New Advertising Campaign Highlighting The Dangers Of Tesla's Full Self-Driving*. 2022. URL: https://dawnproject.com/the-dawn-projects-new-advertising-campaign-highlighting-the-dangers-of-teslas-full-self-driving/.

[44] R. Arokia Paul Rajan. 'Serverless Architecture - A Revolution in Cloud Computing'. In: *2018 Tenth International Conference on Advanced Computing (ICoAC)*. 2018, pp. 88–93. DOI: 10.1109/ICoAC44903.2018.8939081.

[45] Inc. Red Hat. *What is a CI/CD pipeline?* 2022. URL: https://www.redhat.com/en/topics/devops/what-cicd-pipeline.

[46] Inc. Red Hat. *What is CI/CD?* 2022. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[47] Jo Rhett. *Learning Puppet 4: A guide to configuration management and automation.* " O'Reilly Media, Inc.", 2016.

[48] Mojtaba Shahin, Muhammad Ali Babar and Liming Zhu. 'Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices'. In: *IEEE Access* 5 (2017), pp. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.

[49] Muddsair Sharif, Skowronek Janto and Gero Lueckemeyer. 'Coaas: Continuous integration and delivery framework for hpc using gitlab-runner'. In: *Proceedings of the 2020 4th International Conference on Big Data and Internet of Things*. 2020, pp. 54–58.

[50] Kirill Shirinkin. *Getting Started with Terraform*. Packt Publishing Ltd, 2017.

[51] Marcio Silva et al. 'Cloudbench: Experiment automation for cloud environments'. In: *2013 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2013, pp. 302–311.

[52] Charanjot Singh et al. 'Comparison of Different CI/CD Tools Integrated with Cloud Platform'. In: *2019 9th International Conference on Cloud Computing, Data Science Engineering (Confluence)*. 2019, pp. 7–12. DOI: 10.1109/CONFLUENCE.2019.8776985.

[53] Vikas Singh. 'Developing a CI/CD pipeline with GitLab'. In: (2022).

[54] Mark Stillwell and Jose GF Coutinho. 'A DevOps approach to integration of software components in an EU research project'. In: *Proceedings of the 1st International Workshop on Quality-Aware DevOps*. 2015, pp. 1–6.

[55] John S Tonello. 'Automate System Deployments with Terraform'. In: *Practical Linux DevOps: Building a Linux Lab for Modern Software Development*. Springer, 2022, pp. 311–327.

[56] Axel Wikström et al. 'Benefits and challenges of continuous integration and delivery: A case study'. In: (2019).

[57] Yuxia Zhang et al. 'Companies' Participation in OSS Development–An Empirical Study of OpenStack'. In: *IEEE Transactions on Software Engineering* 47.10 (2021), pp. 2242–2259. DOI: 10.1109/TSE.2019.2946156.

This image was generated by Andreas with the assistance of Midjourney [35].