TU WIEN Informatics

Software testing and test environment simulation in the cloud

A case study on open-source IaC and CI/CD technologies

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Emir Halilcevic, Bsc
Matrikelnummer 12206668

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Stefan Biffl

Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn Dietmar Winkler

Wien, 10. Jänner 2025

Emir Halilcevic	Stefan Biffl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Software testing and test environment simulation in the cloud

A case study on open-source IaC and CI/CD technologies

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Emir Halilcevic, Bsc
Registration Number 12206668

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Stefan Biffl

Assistance: Univ.Lektor Dipl.-Ing. Dr.techn Dietmar Winkler

Vienna, January 10, 2025

Emir Halilcevic | Stefan Biffl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Declaration of Authorship

Emir Halilcevic, Bsc

I hereby declare that I have written this Master Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.
I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix "Overview of Generative AI Tools Used" I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, January 10, 2025

Emir Halilcevic

Acknowledgements

First and foremost, none of this would be possible without the relentless support of Dietmar Winkler who believed in the topic from its inception all the way to what it is today, who alongside Prof. Stefan Biffl gave me an opportunity to pursue a topic of my choice, and for that I thank you. Studying at TU taught me that we have the means to teach ourselves much more than we think we are capable of. I also want to thank Hitachi Rail, and more concretely my mentor Peter for offering to support the topic and guiding it to eventually become something more than just a thesis. Also a big thank you to Center for Digital Production for their interest in the topic as well as use case discussion meetings. I want to thank prof. Melisa for believing in me even when I myself didn't, and prof. Amer for making me hold myself to a higher standard and not accept mediocrity.

Throughout my academic journey I always had my family guiding and supporting me and I want to thank them for it. My mom Jasmina for showing me the importance of education, my dad Senaid for teaching me how to bring stability to life through planning, and my sister Amila for always being the older sister to look up to. My grandpa Senadin showed me that age is just a number when it comes to knowledge, my grandma Nura for being my teacher way before I knew what school was. My late grandpa Sakib taught me the importance of friendships, and my late grandma Nevresa showed me the true meaning of the word warrior. My aunt Melika taught me what it really meant to be resourceful and capable, and Sabina who taught me how small the world really is when one seizes the opportunities the world offers.

My family stretched way beyond those I'm related to. My dear friends Azur, Adin, Ismar, and Nermin for the countless jokes that made university life much more tolerable. My friend Gallus for being what we all could strive to be, an honest friend above all. My friend Dino for inspiring me to pursue a topic way out of my comfort zone, and teaching me the importance of patience when dealing with difficulties. My friends Igor, Viktor, Luka, and Benjamin, for making the cold windy Vienna that much warmer and friendlier through countless dorm adventures, while helping me acclimate to a new and foreign world. I want to thank my girlfriend Emina for her endless support and care, for reigniting my love for music and traveling, and showing me the importance of perspective and compassion. My friend Zlatan, for being there from the very start and inspiring me with his confidence and hands on approach to everything, and my friend Ajla for letting

me talk her ear off with all the problems in and out of university. My friend Omar who showed me the importance of investing in oneself, and making sure I did. Among others I want to thank Faruk (Fuke), Faruk (Seha), and Nedzad for staying good friends despite the distance that separates us.

I would like to thank Rastko for inspiring me to do better, holding me accountable, and showing me that close friends can be found even in the most unfamiliar of environments. Most importantly, I'm thankful for not only praising good work, but also pointing out when I could have done better or differently. The years we shared at TU, and the countless coffees at 425 will forever hold a dear place in my heart.

I am incredibly blessed to be able to thank so many people who have shaped who I am today, and thanks to God, blessed with health and strength to close another important chapter of my life. For that, I am truly grateful.

Posvećeno mojoj mami, Jasmini

Abstract

Context: Software testing has long been one of the major pillars of software engineering, as it ensures that the solution produces a correct result, reaches an appropriate outcome, or behaves as intended. Advancements made in cloud computing and DevOps have paved the path for new ways of automating the testing process. Given the inevitability of software testing, automating the process can ease the process making it less tedious by removing manual intervention from the process. Cloud computing allows on-demand access to hardware resources which can be utilized to temporarily create a virtual environment where tests can be performed. Infrastructure as code allows automation of infrastructure deployment to the cloud, and software configuration of the same infrastructure. In combination we can create a virtual environment in the cloud, perform tests, and perform a cleanup leaving no residue in terms of resources, all without human intervention.

Motivation: This can prove useful when the development team needs to ensure that the solution will behave correctly in the deployed environment on the client side. We can simulate the client environment by spawning virtual machines, networks, routers, and perform all the needed software configuration by specifying it through code. The execution of this code can be automated in a CI/CD pipeline that will be triggered when changes to the software solution have been introduced, to ensure that those changes will not negatively impact the solution itself.

Goals and methodologies: Open-source community has made this possible and accessible by providing many technologies that are being used by companies around the world that don't want to be constrained by licenses that introduce entry costs. We aim to cover and explain various practices and tools in the open-source community that have enabled testing in the cloud by performing a literature review. To cover the current tools and practices we will perform various interviews with experts that will give insight in to the topic and the tools. As our goal is to also demonstrate the use of these technologies, we will implement a pipeline prototype following the above described scenario of testing in the cloud, which will be applied to a real life use case in order to evaluate this approach. In conclusion we will provide a better understanding of cloud testing capabilities, while outlining benefits and challenges.

Results: We successfully developed a prototype and applied it to both a fictional use case (server-client communication) as well as a real life use case (Hitachi Rail, Moving Block System) in order to evaluate it. We found that automatic cloud tests improve upon

various areas (e.g. lower testing effort, test usage accessibility, etc.), however they also introduce both a setup complexity overhead as well as a higher bar of entry in terms of both knowledge and additional hardware resources. Most importantly, we outline benefits such as much lower effort needed to use previously manual tests that can be automated, as well as idempotency that can be ensured by always redeploying and destroying the infrastructure needed for testing.

Conclusion: Migration to the cloud offers many benefits (e.g. arbitrary amount of concurrent testing environments) but also drawbacks (e.g. knowledge and resources needed for hosting the cloud) that have to be considered when deciding on it. Despite this, performing tests in the cloud offers much potential and opens door to more possibilities of automating the testing process as well as test environment simulation.

Takeaway: Cloud testing deserves consideration when looking for ways to automate the testing process, but the migration process and resources needed must be taken into account when deciding for this approach.

Keywords: software testing, cloud computing, infrastructure as code, CI/CD, open-source

Kurzfassung

Zusammenhang: Softwaretests sind seit langem eine der wichtigsten Säulen der Softwareentwicklung, da sie sicherstellen, dass die Lösung ein korrektes Ergebnis liefert und sich wie beabsichtigt verhält. Fortschritte im Cloud Computing und DevOps haben den Weg für neue Möglichkeiten zur Automatisierung des Testprozesses geebnet. Angesichts der Unvermeidlichkeit von Softwaretests kann durch Automatisierung der Prozess erleichtert und weniger mühsam gemacht werden, indem manuelle Eingriffe aus dem Prozess entfernt werden. Cloud Computing ermöglicht On-Demand-Zugriff auf Hardwareressourcen, die genutzt werden können, um vorübergehend eine virtuelle Umgebung zu erstellen, in der Tests durchgeführt werden können. Infrastruktur als Code ermöglicht die Automatisierung der Infrastrukturbereitstellung in der Cloud und der Softwarekonfiguration derselben Infrastruktur. In Kombination können wir eine virtuelle Umgebung in der Cloud erstellen, Tests durchführen und eine Bereinigung durchführen, die keine Ressourcenrückstände hinterlässt, und das alles ohne menschliches Eingreifen.

Motivation: Dies kann sich als nützlich erweisen, wenn das Entwicklungsteam sicherstellen muss, dass sich die Lösung in der bereitgestellten Umgebung auf der Clientseite korrekt verhält. Wir können die Clientumgebung simulieren, indem wir virtuelle Maschinen, Netzwerke und Router erstellen und alle erforderlichen Softwarekonfigurationen durchführen, indem wir sie durch Code angeben. Die Ausführung dieses Codes kann in einer CI/CD-Pipeline automatisiert werden, die ausgelöst wird, wenn Änderungen an der Softwarelösung vorgenommen wurden, um sicherzustellen, dass diese Änderungen die Lösung selbst nicht negativ beeinflussen.

Ziele und Methodiken: Die Open-Source-Community hat dies möglich und zugänglich gemacht, indem sie viele Technologien bereitstellt, die von Unternehmen auf der ganzen Welt ohne Einstiegskosten und Lizenzen verwendet werden können. Wir möchten verschiedene Praktiken und Tools in der Open-Source-Community abdecken und erklären, die Tests in der Cloud ermöglicht haben. Um die aktuellen Tools und Praktiken abzudecken, werden wir verschiedene Interviews mit Experten durchführen, die Einblicke in das Thema und die Tools geben. Da es unser Ziel ist, auch die Verwendung dieser Technologien zu demonstrieren, werden wir einen Pipeline-Prototyp gemäß dem oben beschriebenen Szenario des Testens in der Cloud implementieren, der auf einen realen Anwendungsfall angewendet wird, um diesen Ansatz zu bewerten. Abschließend werden

wir ein besseres Verständnis der Cloud-Testfunktionen vermitteln und gleichzeitig Vorteile und Herausforderungen skizzieren.

Ergebnisse: Wir haben erfolgreich einen Prototyp entwickelt und ihn sowohl auf einen fiktiven (Server-Client-Kommunikation) als auch auf einen realen Anwendungsfall (Hitachi Rail, Moving Block System) angewendet, um ihn zu bewerten. Wir haben festgestellt, dass automatische Cloud-Tests in verschiedenen Bereichen Verbesserungen bringen (z. B. geringerer Testaufwand, Zugänglichkeit der Testnutzung usw.), jedoch auch einen höheren Aufwand bei der Einrichtung sowie höhere Einstiegshürden in Bezug auf Wissen und zusätzliche Hardwareressourcen mit sich bringen. Am wichtigsten sind die Vorteile, die wir skizzieren, wie den viel geringeren Aufwand bei der Verwendung zuvor manueller Tests, die automatisiert werden können, sowie die Idempotenz, die durch die ständige Neubereitstellung und Zerstörung der für Tests erforderlichen Infrastruktur sichergestellt werden kann.

Fazit: Die Migration in die Cloud bietet viele Vorteile (z. B. beliebige Anzahl gleichzeitiger Testumgebungen), aber auch Nachteile (z. B. Wissen und Ressourcen, die zum Hosten der Cloud erforderlich sind), die bei der Entscheidung darüber berücksichtigt werden müssen. Trotzdem bietet die Durchführung von Tests in der Cloud viel Potenzial und öffnet die Tür zu mehr Möglichkeiten der Automatisierung des Testprozesses sowie der Simulation von Testumgebungen.

Takeaway: Cloud-Tests verdienen Beachtung, wenn nach Möglichkeiten zur Automatisierung des Testprozesses gesucht wird. Bei der Entscheidung für diesen Ansatz müssen jedoch der Migrationsprozess und die benötigten Ressourcen berücksichtigt werden.

Schlüsselwörter: Softwaretests, Cloud Computing, Infrastruktur als Code, CI/CD, Open-Source

Contents

CHAPTER 1

Introduction

1.1 Motivation

In the domain of software development, testing is one of the ways of uncovering errors in the software solution [73]. This, often done in the developer environment, on the developer's machine, sometimes leads to problems when the solution behaves differently in the client environment, client's machine, where previously passing tests start to fail, or the solution fails to run. If the client configuration is known, one can simulate said environment by configuring all the required software and using all the required hardware resources used by the client. This means that further costs are introduced in the form of physical hardware needed but also additional time needed to setup and maintain this environment.

Any changes made in the client configurations would have to be reflected on the simulated physical one used for testing. This can prove tedious and costly depending on how dynamic the client configuration is. Instead of physically trying to imitate an environment, one could create a virtual one in the cloud, perform tests and report back the results. This can be achieved through an automatic pipeline where we could through code specify the client configuration. We can achieve this through the concept of Infrastructure as Code (IaC).

IaC along with Continuous Integration/Continuous Deployment (CI/CD) have spawned many new technologies that allow us to make a fully automatic end-to-end pipeline that ensures the quality and consistency of the software solution from build to deployment without the need for manual intervention. By testing in the cloud one can utilize both a privately hosted in-house cloud or use an on-demand public cloud to leverage the pay-per-use model.

In their work [70] Mittal et. al. argue for cloud testing as the future of software testing by outlining the types of existing cloud testing techniques as well as their needs and

benefits in form of scalability, adaptability and cost. By simulating the client environment in various cases and performing various tests, the goal is to improve efficiency of the testing process by avoiding manual setup and configuration, and avoiding costs of physical machines by utilizing the cloud which can be in-house or remote using pay-per-use model. By automatizing this testing process through the use of a CI/CD pipeline, less time is needed to adapt to client changes resulting in better client satisfaction.

1.2 Problem statement

Let us assume a team of developers are working on developing a software solution that will be delivered to a client (Figure 1.1). This client has clearly specified the resources (e.g. amount of RAM) and the software (e.g. operating system) of the machine on top of which the software solution will run. The developers are tasked with ensuring that the developed software solution runs correctly on the client machine. Unfortunately they do not have access to the client environment/machine, nor do they posses the resources nor the software used by the client.

The developers' and the client's environment are in no way similar. One approach that the team takes is they order the specific hardware components for a physical machine that they will maintain manually in- house. Following this, they install the required operating system, and dependent software/libraries which are needed for the solution. The testers connect to this machine, manually install the software and perform all the needed tests themselves. Once all tests pass, the client receives the software solution.

1.2.1 Stakeholders

The following stakeholders and their goals can be identified:

- **The developer** wants to immediately know if the changes they have performed on the software solution cause any problems should the software solution be run on the client machine, and be provided with exact results on which test passes or fails.
- **The tester** wants to be able to describe the client environment through the use of IaC configuration files that specify hardware resources and the software needed, and write tests that will be performed on the previously described environment every time changes are made to the software solution.
- **The client** wants the software solution to run correctly on their machine/environment, and wants to be able to change his hardware resources and software, communicate it to the developers, and be quickly insured that changes on the client side will not affect software solution correctness.
- **The DevOps engineer** wants to adapt the devops lifecycle so that any and all client configuration changes are quickly adapted to.

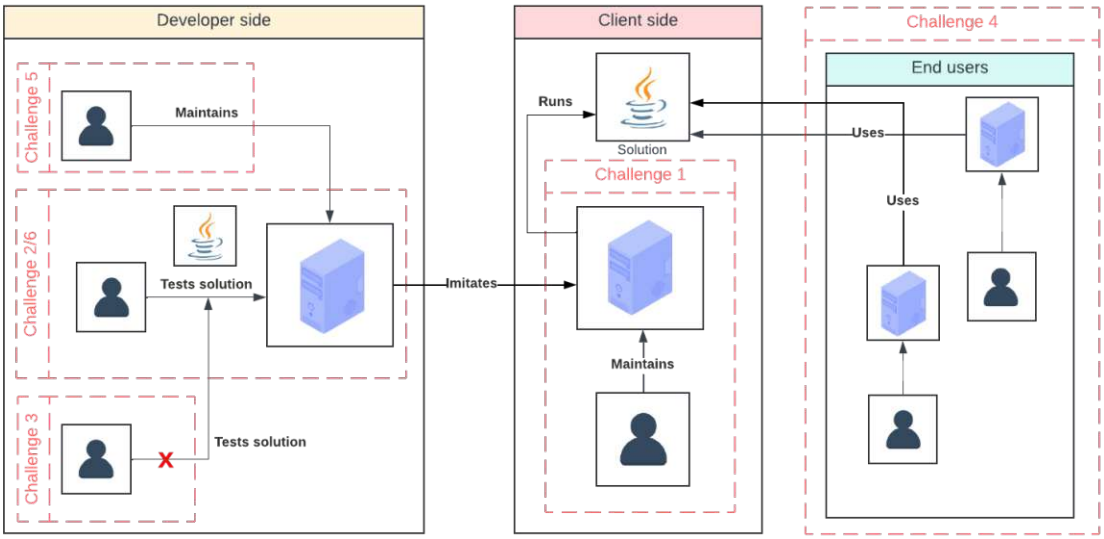

Figure 1.1: High-level overview of the example problem

1.2.2 Challenges

By analyzing this example we can identify the following challenges that may arise during the development cycle:

- **Challenge 1: Adapting to changes.** The client has changed the software (e.g. operating system) on top of which the software solution will run, and/or they have increased or decreased hardware resources available for running the software solution. The software solution needs to be tested against these changes which requires manual maintenance of the testing environment (acquiring/installing new software, adjusting hardware resources, etc.) to exactly match the client setting.

- **Challenge 2: Ensuring test idempotency.** Between each test, the test environment must be identical, and must not be affected through successful or unsuccessful test runs, which should insure test idempotency.

- **Challenge 3: Enabling concurrent test environment usage.** For concurrent usage of the test environment, every tester needs to have their own environment which in the case of physical machines requires a machine (or machines) per each person, introducing further costs.

- **Challenge 4: Simulating the end users.** In case of a software solution that interacts with other users, these users also have to be simulated, and their interaction with the solution.

- **Challenge 5: Maintaining the test machine.** Manually maintaining and configuring a physical machine to match that of the client, requires additional effort.
- **Challenge 6: Removing human intervention.** Human intervention is needed throughout the testing process which could be avoided through automation based on CI/CD principles, the cloud and IaC technologies.

Ideally, should resources not be a limitation, having a dedicated developer who, given the exact client machine/environment specification, creates a physical machine with exact hardware and software, would prove to be the most physically accurate representation of the client side. However, as the configuration process is manual, any and all changes have to me manually tended to, which slows down the process of adapting to changes. Should more than one machine be required, the network of the machines has to also be physically identical.

By utilizing IaC and the cloud, we can create a virtual environment for various types of tests tailored specifically for each of them, where hardware resources and software can be configured through code, avoiding any manual installation, while also allowing to create many virtual environments suited for various different client specifications.

1.3 Goals

The objective of this work is to explore, research, and apply various open-source technologies, while providing a hands-on approach to implementing a general purpose end-to-end testing pipeline. This work aims to leverage cloud for the purposes of creating test environments in order to provide idempotent and replicable virtual environments for solution testing. The aim is to assess the current capabilities of various open-source technologies and their interaction in order to support CI/CD principles and ensure thorough testing in the cloud.

We will outline the importance of DevOps practices such as Infrastructure as Code, and Continuous Integration/Continuous Deployment, followed by an analysis of various open-source technologies that support them such as Gitlab CI, Ansible, Chef, Terraform and similar. We will then take a look at how we can use cloud computing to our benefit in order to emulate various components of the client environment such as hardware setup and various hardware components.

Following this, we will clearly define a set of requirements that the pipeline needs to fulfill to achieve our goal of testing in the cloud using simulated environments. In order to decide between the technologies, besides the requirements, we will conduct interviews with various DevOps, infrastructure and cloud computing experts which will serve as basis for deciding on a technology stack that will practically demonstrate the pipeline.

This will serve as the foundation for building our prototype that will later be applied to a use case through which we will evaluate the approach, by performing a cost benefit

evaluation. We will compare the benefits of each approach (cloud testing and traditional testing) and compare the resources needed for each approach in terms of both effort and time.

The main objective of the work is to utilize open-source CI/CD technologies to automate the testing process by developing a pipeline that will utilize open-source IaC technologies to perform tests in the cloud.

Above described goals can be summarized through the following list:

- **G.1 (Literature review)**: Research, understand and explain state-of-the-art technologies and approaches that exist, and how they enable cloud testing.
- **G.2 (Pipeline requirements elicitation)**: Understand and define what the pipeline needs to do, while addressing all the challenges defined in Section 1.2.2.
- **G.3 (Experts' insight)**: Elicit knowledge and insight from experts and practitioners to better understand how to implement the pipeline and what potential benefits and challenges cloud testing offers.
- **G.4 (Implement pipeline prototype)**: Implement the pipeline using a defined technology stack of state-of-the-art open source technologies.
- **G.5 (Benefits, challenges, drawbacks)**: Understand what benefits cloud testing provides in comparison to the traditional testing methods. Outline challenges and drawbacks of cloud testing.

1.4 Structure

The work follows the following structure:

Chapter 2 provides background knowledge needed for understanding the context of the work, like DevOps, CI/CD, cloud computing and IaC. An overview of state-of-the-art technologies is provided, followed by a literature review of related work.

Chapter 3 introduces the research questions that address the goals defined in the previous Section 1.3. An explanation is provided as to how each research question will be addressed.

Chapter 4 provides a comprehensive list of requirements that the pipeline needs to fulfill based on user stories and challenges introduced in the Subsection 1.2.2. This will serve as the basis for the pipeline implementation.

In Chapter 5 we conduct interviews with various experts and practitioners in order to gather insight and knowledge in to what open-source technologies to utilize during the implementation. Through the interviews we also get a better understanding of benefits, drawbacks of testing in the clouds, as well as types of tests worth exploring. At the end of the chapter we provide our findings

Chapter 6 covers the implementation of the pipeline prototype, explaining in detail how each technology comes in to play in order to achieve automation. We introduce an example problem, followed by an explanation of what type of test we want to simulate in the cloud and then implement it by deploying a simulated environment through the use of the pipeline.

Chapter 7 applies and adapts the prototype pipeline to a real life use case. The use case will be in the railway domain applied to a project at Hitachi Rail. We will take a look at what improvements the pipeline makes by analyzing and comparing resources in terms of time and effort needed for both approaches.

In Chapter 8 we will provide and discuss the answers to the research questions. This will be followed by a discussion on limitations and challenges.

Finally, Chapter 9 will conclude the work by summarizing the results of the work and provide avenues for future work.

CHAPTER 2

Related work

2.1 Software engineering best practices

Unit tests are one of the most well established testing methodologies for testing the components of the software solution in isolation [77]. Integration tests on the other hand aim to test the interaction between many components [67]. Unlike unit tests, integration tests are sometimes harder to define through code, so in some cases developers choose to perform them manually by running all the necessary components, simulating a use case and acting as the test oracle. This can be automated through various tools, but the solution components run in the developer's environment limited by the developer's machine. We will now take a look at various methodologies and tools that have been developed, which will help us move the testing process (e.g. integration tests) to the cloud, decoupling tests from the developers environment.

We will take a look at various practices and methodologies that aim to improve some aspect of the software development cycle. We will take a look at DevOps, followed by various DevOps practices and tools, more concretely CI/CD and IaC. We will conclude the best practices by covering cloud computing, which is an important aspect of this work. This will be followed by a literature review that will present various papers that cover the topic relevant to this work.

2.1.1 DevOps

DevOps, a name resulting from the combination of two words, development and operations, has seen many definitions over the years. It is an integration of both worlds through the use of automated development, deployment, and infrastructure monitoring [56]. DevOps can also be defined as effort to automate continuous delivery while guaranteeing correctness [66]. Regardless of how one defines it, what DevOps aims to be is a bridge between development and operations while facilitating communication. DevOps also

encompasses various tools and practices that aim to aid and improve the synergy between the two worlds. We will now take a closer look at one of the major pillars of the DevOps practices.

2.2 Continuous Integration and Continuous Delivery/Deployment

As software development matured, and the need for faster integration of changes to the software solution grew, so have various methodologies and principles that aim to speed up the development cycle.

Continuous Integration, as defined by Fowler in their work [57], represents a software development practice that aims to speed up integration of new work, where an automated build verifies it. They go on to introduce various practices of CI, amongst others, build automation, self-testing builds and production environment clone testing. They outline virtualization as the key to assembling a test environment that, to a certain extent, mimics that of the client. Much of what they described, like single source repository, would now be considered standard practice, as more and more projects reap the benefits of CI, which has spawned various tools that aim to support its practices. Although the concept of CI, has existed before the publishing of Fowler's work, it certainly sparked interest in CI practices. In hopes of shedding new light on benefits of CI, Hilton et al. [63] perform a study on the usage of CI in open-source projects, and have found that CI does actually facilitate more frequent releases.

Continuous delivery continues this automation quest where after code changes have been tested, the solution itself is deployed to some non-production environment for testing or staging [44], in order to ensure that it can always be reliably released [52]. Although not always possible, by also automating the deployment to the production environment, we would be practicing continuous deployment, often confused with continuous delivery as they are commonly abbreviated as just CD and used as synonyms [78]. In result we have CI/CD as the umbrella term that groups together the automation principles from code development to deployment.

All three phases have their own set of benefits and challenges [57, 52, 78], but provide guidelines for encouraging automation of the development cycle, while spawning many tools that utilize CI/CD.

2.2.1 Gitlab CI

In 2015 Gitlab CI got integrated into Gitlab [64], while existing before as a standalone application. It represents an automation tool that enables implementation of various CI/CD principles. Gitlab CI (CI/CD)[12] allows development of a pipeline within a code repository on Gitlab. Using YAML syntax, a pipeline is described in `.gitlab-ci.yml` file (or several files). A Gitlab CI pipeline consists of stages that group together jobs that perform certain actions on the code in the repository. A usual workflow consists of building the code, testing, and deploying. Gitlab Runner [14] is a separate application hosted independently which runs the jobs in the pipeline. Following is an example of a simple pipeline that performs the above described workflow:

```
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - mvn build

test:
  stage: test
  script:
    - mvn test

deploy:
  stage: deploy
  script:
    - mvn deploy
```

Listing 2.1: Gitlab CI pipeline example

2.2.2 Jenkins

Released in 2011, Jenkins [20] is an automation platform that aims to automate various parts of the development cycle. Unlike Gitlab CI, it supports many other functionalities besides pipeline definition, and is not tied to any concrete repository manager. Jenkins agents (similarly to Gitlab runners) perform tasks from the Jenkins controller which is the Jenkin service itself [42].

Jenkins allows the definition of the pipeline in two ways usually stored as `Jenkinsfile`, declarative pipeline and scripted pipeline, with scripted pipeline providing more freedom, but having a steeper learning curve compared to more user friendly declarative pipeline. The scripted pipeline definition uses a limited Groovy syntax [28], while the declarative pipeline offers a higher level syntax that abstracts away some details. An example of a declarative pipeline is as follows:

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'mvn build'
            }
        }

        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }

        stage('Deploy') {
            steps {
                sh 'mvn deploy'
            }
        }
    }
}
```

Listing 2.2: Jenkins pipeline example

One of the most popular Jenkins features is the ability to introduce add-ons to Jenkins through the concept of Jenkins plugins, but introduces maintenance overhead as these plugins are developed separately from the Jenkins.

2.3 Infrastructure as Code

The need for constant manual setup, configuration and upkeep of machines, virtual or physical, has paved the road for tools that would automate this process, especially needed in situations of a large number of machines. Infrastructure as code (IaC) aims to eliminate manual infrastructure management through automation, which supports the principles of CI/CD (Section 2.2) and subsequently DevOps (Section 2.1.1).

2.3.1 Configuration management

IaC tools have been in development for a long time, and can be traced back to CFEngine developed by Mark Burgess in 1993 for the University of Oslo. In their work [50], they describe a high level description language for machine setup and configuration. Through classes and actions per class one can define the needed steps to be performed on the configured machine. In essence CFEngine represents a prime example of configuration management tool where through a program file one can configure several machines (install

software, run services, etc.). This would later inspire other emerging tools to improve upon the foundation that CFEngine laid, such as Puppet (2005), Chef (2009), and Ansible (2012), with each improving some aspect over the previous.

2.3.2 Puppet

Developed by Puppet Inc. in 2005, Puppet [33] is a configuration management tool for automation of server configuration, and is one of the earliest examples of of widely adopted configuration management through IaC. It uses its own declarative style language to define some desired state of the client, using a concept of Catalogs [34]. Using Resources [34] the developer can describe some requirements (e.g. ensure package curl is present Listing 2.3), that can be grouped in to Classes [34], all contained within a file called Manifest [34] (similar to Ansible playbook, or Chef recipe). It follows a server-client approach, requiring the managed client to have an agent installed. For example, a Resource that ensures the presence of package curl, and installs it otherwise is presented as follows:

```
1 package { 'curl':
2   ensure => installed,
3 }
```

Listing 2.3: Puppet IaC example

2.3.3 Chef

Chef [5] (also known as Progress Chef), and more precisely Chef Infra as one of Chef tools, is a configuration management framework developed by Adam Jacob and released in 2009. Chef architecture [4] follows an agent-based approach where each managed node (machine) needs to have Chef Infra Client installed which pulls any changes that need to take place from the Chef Infra Server running separately. The developer uses Chef Workstation to describe the configuration through Ruby code in units called recipes [2] grouped together into cookbooks [1] that are pushed to the Chef Infra Server. Chef does offer a serverless approach, but it still requires client installation. A simple recipe that ensures that curl is installed can be written as follows:

```
1 package 'curl' do
2   action :install
3 end
```

Listing 2.4: Chef IaC example

2.3.4 Ansible

Developed by Michael DeHaan in 2012 [65], Ansible [15] is a IaC tool for configuration management. It enables developers to, through code, specify the desired state of the machine in terms of software and configuration with Ansible ensuring the target machine

achieves it. The machine running Ansible is considered the control node and the target machine(s) is considered the managed node(s)[45]. Ansible uses the concept of a playbook to decleratively define tasks that have to be performed on the target machine. These tasks contain modules [3] that perform actual commands on the controlled node. For example an Ansible playbook containing one task that ensures that curl is installed, and if not installs it:

```
- name: Setup server
  hosts: server
  gather_facts: no
  become: true
  tasks:
    - name: Ensure curl is installed
      apt:
        name: curl
        state: present
```

Listing 2.5: Ansible playbook example

Ansible manages nodes by grouping them into an inventory [16] that can be specified by the user as a file supporting INI and YAML syntax. A network containing two managed nodes along login credentials would be configured as:

```
[network1]
server ansible_host=1.1.1.11 ansible_user=ubuntu
    ansible_private_key_file=~/.ssh/id_rsa/id_rsa
client ansible_host=1.1.1.10 ansible_user=ubuntu
    ansible_private_key_file=~/.ssh/id_rsa/id_rsa
```

Listing 2.6: Ansible inventory example

What makes Ansible stand out compared to the Chef and Puppet is its agentless approach. However, Ansible does impose requirements on the managed nodes, requiring them to support SSH and have Python installed. Ansible is attributed with easy readability as execution of task is performed in step-by-step manner described using YAML syntax.

2.3.5 Provisioning

Provisioning can be defined as one step before configuration management, tasked with the actual creation of the infrastructure, whereas configuration management is tasked with installing and configuring software on already existing infrastructure. Robust IaC tools often carry the ability of both configuration and provisioning (e.g. Ansible os_server module [26]), which sometimes blurs the line between the two. Usually IaC tools tend to be specialized for either provisioning or configuration management resulting in two tools working in tandem. One of the most popular examples would be Terraform [39] introduced in 2014 by HashiCorp, where one can define all the needed infrastructure (virtual machines, subnets, routers, etc.) through a Terraform defined language, where

the interaction with the cloud is abstracted away through a concept of providers [29]. The importance of provisioning has motivated the introduction of various other provisioning tools like Pulumi (2017), Crossplane (2018), and OpenTofu (2023). Provisioning could be categorized based on the type of infrastructure that is being provisioned, but for the purposes of this work we will focus on cloud provisioning of cloud infrastructure.

2.3.6 Terraform

Developed by HashiCorp in 2014, Terraform [39] is a IaC tool for infrastructure provisioning specified through code in declerative fashion using Hashicorp Configuration Language [38]. Terraform abstracts the interaction with the cloud through the use of providers [29], that are separately developed for each cloud provider, allowing the user to more easily change their cloud provider while maintaining the same provisioning tool.

Using the OpenStack provider [40] one can define a virtual machine attached to a network, specify the hardware through the concept of flavors, and the image for running the machine. The following defines a virtual machine with 1 virtual cpu, 1GB of RAM, and 5GB of storage running Ubuntu 24.04 image:

```
resource "openstack_compute_instance_v2" "vm" {
  name            = "vm"
  image_name      = "Ubuntu 24.04 LTS"
  flavor_name     = "general-1C-1G-5GB"
  key_pair        = "id_rsa"
  security_groups = ["all-icmp", "ssh+https"]

  network {
    uuid = "aaa-bbb-ccc"
  }
}
```

Listing 2.7: Instance definition using Terraform

Terraform also allows previewing the changes that will be made in the cloud through the concept of planning [8], before applying [6] the changes. Terraform keeps track of the infrastructure state [37] stored as a file that binds the configuration to the deployed infrastructure. Using this state, Terraform can very easily cleanup and destroy [7] the spawned infrastructure, which would prove very useful when idempotency is a must.

2.3.7 Is Terraform open-source?

On 10th of August 2023 HashiCorp announced a change in their license [53] as HashiCorp would switch from the Mozilla Public License (MPL) v2.0 to Business Source License (BSL/BUSL) v1.1. MPL v2.0 [22] is a copyleft license and is an open source license, and as such encourages the use of code for any purpose, in contrast to BSL v1.1 which puts additional restrictions on what use is considered appropriate use within the license. Hashicorp states better management of commercial use of their source code as the reason

for this change, and using BSL v1.1 defines "competitive offering" of their code as in breach of the license [62]. However, HashiCorp-built providers for Terraform are still under MPL 2.0 license, and the acquisition of HashiCorp by IBM may also provide hope of HashiCorp returning back to MPL v2.0 [59].

This effectively means that Terraform is no longer open-source despite the source code being freely accessible [13], a lot is left ambiguous as to who may be considered competition in the eyes of HashiCorp. With no guarantees that Terraform won't restrict the license even further, and with the focus of this work being on open-source technologies, Terraform will not be considered as one of the open-source technologies for the development of the pipeline.

2.3.8 OpenTofu

As a solution to this we may consider an open-source Terraform-based alternative. OpenTofu [21] is a fork of the last MPL licensed Terraform repository, backed and maintained by the Linux Foundation, which aims to offer an open-source alternative to Terraform. This means that from this point on Terraform and OpenTofu will diverge, however since the fork has happened rather recently, coupled with the fact that Terraform providers are still open-source, the transition from Terraform and OpenTofu at this point in time is almost seamless. Another approach would be to only use Terraform versions that are older than August of 2023, but this doesn't align with the goal of this work to explore the state-of-the-art technologies.

2.3.9 Pulumi

Pulumi [30], developed by the company of the same name by Joe Duffy and Eric Rudder in 2017 [49], is an IaC tool for cloud provisioning automation. It allows its user to define the infrastructure through various programming languages, such as Python or Go, while still offering the use of markup languages such as YAML. Using one of them, the user writes Pulumi programs and describes the desired infrastructure through the concept of resource object [9]. Pulumi programs are grouped together in a project. Similarly to Terraform, Pulumi allows a preview of changes that will take place by calling Pulumi preview [31], followed by Pulumi up [32] that will instantiate the project as a stack [36].

```
resources:
  vm:
    type: openstack:compute:Instance
    properties:
      name: "vm"
      imageName: "Ubuntu 24.04 LTS"
      flavorName: "general-1C-1G-5GB"
      keyPair: "id_rsa"
      securityGroups:
        - "all-icmp"
        - "ssh+https"
```

```
12        networks:
13          - uuid: "aaa-bbb-ccc"
```

Listing 2.8: Instance definition using Pulumi

2.4 Cloud computing

Similarly to DevOps, the term cloud computing envelops many different definitions that have been proposed throughout the years [85]. At its core, cloud computing represents a way of distributing computing resources (hardware, services) to the user in an on-demand manner. From the perspective of the user, these resources are abstracted and virtualized which enables the user to request and pay for resources as needed. This of course shifts the responsibility of maintenance to the cloud provider who may be public, or a private on-premise cloud, but a combination of both is possible. Depending on what resources are provided as a service, different cloud computing models have been developed [43]:

- IaaS (Infrastructure as a Service) for on demand servers, storage, and networking
- PaaS (Platform as a Service) for on demand platform/environment
- SaaS (Software as a service) on demand applications

DevOps and cloud computing in combination offer many capabilities of automatizing different aspects of the development cycle. This integration of practices and technologies has found its ways in various real world applications due to its ability to streamline application delivery [55], arguing for its future potential to influence a shift toward a more automatized development cycles.

2.4.1 OpenStack

OpenStack [24], developed by Rackspace and Anso Labs in 2010 [19], is an open-source cloud computing platform for management of compute, storage and network resources. OpenStack pools together hardware resources and makes them available to the user in an API driven way. OpenStack is modular in the sense that functionalities are extracted in to services that interact with eachother through public APIs. In essence, this means that an administrator may choose a subset of services that they actually need. Of most relevance to the work are the following services:

- **Horizon** brings the functionality of a dashboard, or in other words a GUI for a more visual interaction with OpenStack. It is however not required and OpenStackClient [25] can be used for a CLI interface.
- **Keystone** implements OpenStack's Identity API [17], and is used for authentication and authorization.

- **Nova** offers the ability to provision compute instances, like spawning of virtual machines. It allows configuration of hardware resources through the concept of flavors (vCPUs, RAM, storage).
- **Glance** manages virtual machine images.
- **Neutron** implements the OpenStack Networking API [23] and provides network capabilities like provisioning of networks, routers, subnets, and more. It enables the concept of software defined networks.

2.5 Related projects and publications

A systematic mapping study of infrastructure as code research [76] at the time showed that most research in IaC focused on implementing the practice of IaC, with the most used tool being Chef. It also outlines the importance of the topic for the field of software engineering while also identifying various potential research avenues that could be explored like what the challenges in learning and implementing IaC are.

In their work [70] Mittal et. al. argue for cloud testing as the future of software testing by outlining the types of existing cloud testing techniques as well as their needs and benefits in form of scalability, adaptability and cost. In the work by Thatikonda [83], CI/CD best practices and principles are explored and two IaC tools are outlined, Terraform for its platform-agnostic nature and Ansible for its configuration management capability. A systematic review on Cloud Testing [47] showed that a lot of interest is placed on testing in the cloud, and analyzed six areas of cloud testing research, one of them being test execution where most works presented cloud testing tools but only a few discuss specific testbed setups in cloud environments.

The use of a virtual environment to avoid testing on actual hardware has found it use in various applications. In automotive industry when hardware isn't available virtual environments can be employed to perform tests and diagnostics [86, 48]. When testing mobile applications, one can utilize the cloud to test with real network traffic and to test multiple portable devices in a single system [68, 46]. The work by Mitesh [82] provides a proof of concept on how end-to-end automation could be implemented in the cloud in the insurance industry. An exploration of open-source cloud platforms is provided by Bashir et al. [71], with a proof of concept of a cloud lab for testing and experiments using OpenStack.

In their work, Vogel et al. [84] performed a comparative analysis of various cloud tools like OpenNebula, CloudStack and Openstack. They found OpenStack to be the system most adaptable to changes and failures in order to maintain availability, and in their performance evaluation found workloads running on OpenStack instances to be the most stable. Another paper achieves a similar goal, where Jaison et al. [72] conduct a comparative study between OpenStack and Cloudstack, outlining OpenStack's stability and performance as its benefit, however it also comes with a difficult installation process

compared to CloudStack. Paradowski et al. [75] conducted a performance benchmark between OpenStack and CloudStack on various metrics and concluded that despite both being developed by Citrix, OpenStack outperforms CloudStack in every benchmark criteria.

Gupta et al. [61] practically demonstrated how one can automate the process of Hadoop cluster deployment to AWS cloud using Ansible for configuration management and Terraform for provisioning, with the goal of cluster setup automation without human intervention. While arguing for Terraform, Abbas et al. [69] show how Terraforms usage has risen highly compared to the number of users/organizations, and they emphasize how Terraform demonstrates convenience and consistency when deploying apps on AWS cloud.

A. Dhanapal et al. [54] demonstrated practically how OpenStack can be used for HTTP flooding attack evaluations, by simulating real-time attacks using a cloud environment, resulting in a tesbed framework. In a paper by Sicoe et al. [79], it is examined how one can automate deployment of a network topology using Openstack cloud and Terraform for provisioning. They configure virtual machines on the cloud using Ansible in order to run Cisco routers on top of them.

Singh et al. [80] in their work performed a comparison between Gitlab CI and Jenkins for deployment to AWS cloud, and concluded that it becomes difficult to manage Jenkins in case of many plugin, in contrast to Gitlab CI that configures the pipeline through one YAML file regardless of project size. Using Jenkins and Ansible one can also achieve pipeline automation following CI/CD principles, as explained by Sriniketan et al. [74], where they use Jenkins due to its plugins (e.g. Jenkins Ansible). They also argue for Ansible over Chef and Puppet, due to its agentless approach to configuration management.

Deployment using a fully automated CI/CD pipeline is explored by Singh et al. [81] using Jenkins, Ansible, Kubernetes and AWS, arguing for, amongst other advantages, complete automation and time efficiency. Cepuc et al. [51] take a similar approach by using the same set of technologies to achieve an automatic pipeline for Java-based web application deployment.

In comparison to the aforementioned related work we aim to close the research gap in the domain of open-source by providing a state-of-the-art overview, accompanied by insight from experts, and applied to the railway domain. We aim to provide a hands-on approach to IaC and CI/CD in order to argue for its use for testing in the cloud, and demonstrate its influence when applied to a real-life use case. The goal is to also provide a guide in the form of pipeline requirements and a technology stack derived from interviewing practitioners well versed in CI/CD, IaC, and cloud computing.

CHAPTER 3

Research questions

We will begin by performing a literature review, and researching state-of-the-art technologies, as this will be our first research question RQ1. We will also cover software engineering practices and principles and technologies that enable them. Following this we will perform requirements elicitation to define pipeline requirements as part of RQ2. RQ3 will answer the question of what technology stack to use for the implementation upon which we will perform an evaluation that will answer RQ4.

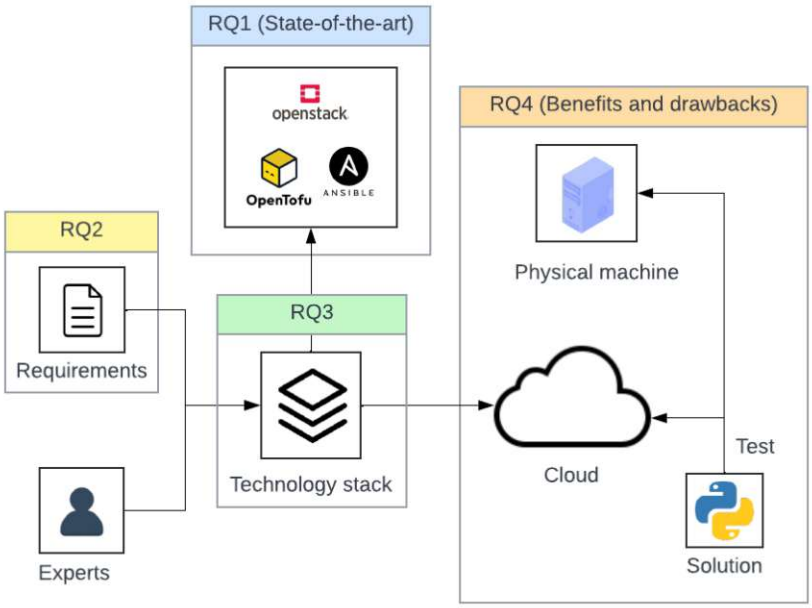

Figure 3.1: Research questions visually represented

In Section 1.3 we defined the goals we aim to achieve in this work, and from these we can derive the following research questions and how we plan to address them:

- **RQ.1: What are the state-of-the-art open-source IaC and CI/CD technologies for enabling testing in the cloud using simulated environments?**

  As defined in goal G.1 we want to provide an overview of technologies and approaches that enable us to perform testing in the cloud using simulated environments. We aim to also explain the background behind them to better understand their role in achieving cloud testing. To address this goal we define RQ.1 that aims to answer this, which requires us to perform a literature review of state-of-the-art IaC, CI/CD, and Cloud computing technologies, and related scientific literature that will give us insight in to various usage of these technologies. Answering these two research questions will also help us better understand what the requirements for the pipeline are, by understanding the capabilities of current CI/CD, IaC, and Cloud computing technologies, which is the main goal of G.2.

- **RQ.2: What are the requirements of a fully automated pipeline for testing in the cloud using simulated environments?**

  Addressing goal G.4 and answering this research question, requires understanding the criteria for the pipeline implementation. We've previously defined in Section 1.2.2 challenges we are trying to address, and stakeholders in Section 1.2.1, which will serve as the foundation for requirements elicitation that will provide as with a requirements specification that we will implement.

- **RQ.3: What is the architecture for a fully automated pipeline for testing in the cloud using simulated environments?**

  In goal G.3 we aim to elicit knowledge and recommendations from experienced experts which will serve as the foundation to answering this research question. To do so, we will perform interviews with a selection of experts and practitioners that will answer various questions related to their experience and knowledge of IaC, CI/CD, and cloud computing technologies. Once concluded, we will be able to recommend a technology stack for achieving the automated pipeline, and as such will directly answer RQ.3. As goal G.5 is to also provide insight into benefits and challenges, one part of the interview will be dedicated to addressing this goal.

- **RQ.4: What are the benefits and drawbacks of testing in the cloud using simulated environments compared to the traditional testing methods?**

  After finalizing the pipeline prototype (goal G.4), we will better understand obstacles and difficulties one might face during the implementation. By addressing goal G.3 as described above, we will also be informed of advantages and disadvantages of cloud testing. Moreover, by performing a use case study, by applying the pipeline prototype to a real-life use case, we will perform a cost benefit evaluation through a comparison applied to this use case directly addressing G.5.

CHAPTER 4

Requirements analysis

4.1 Requirements elicitation

The following chapter aims to clarify and define the exact requirements the pipeline needs to fulfill without focusing on any concrete technology, in essence we present a technology agnostic description of all the needed tasks the pipeline should fulfill. We will define the pipeline requirements with the regards to the actual design of the pipeline. Following this, we will outline the pipeline requirements related to the cloud tests that will be performed in the pipeline. In the end, we will also define requirements for the technologies that will be used for purposes of configuration management and provisioning.

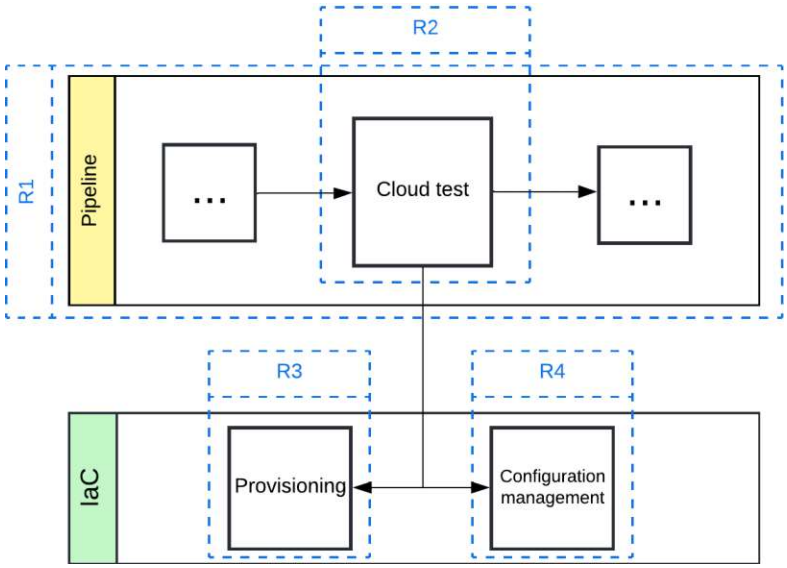

Figure 4.1: Research questions visually represented

The requirements will be divided into four section. We will consider requirements regarding the pipeline design itself, and will be label as R1. This will be followed by requirements that address the structure of tests and will be labeled as R2. In the end we will also define requirements for tools that will be used for configuration management (R3) and provisioning (R4). The classes were created as a way of organizing the requirements as they group together relevant requirements.

In order to elicit the requirements, we will define user stories from a perspective of one of the stakeholders defined in Subsection 1.2.1. Each story will be followed by a requirement derived from it, and should the requirement address a specific challenge defined in Subsection 1.2.2 it will be noted next to the name of the requirement.

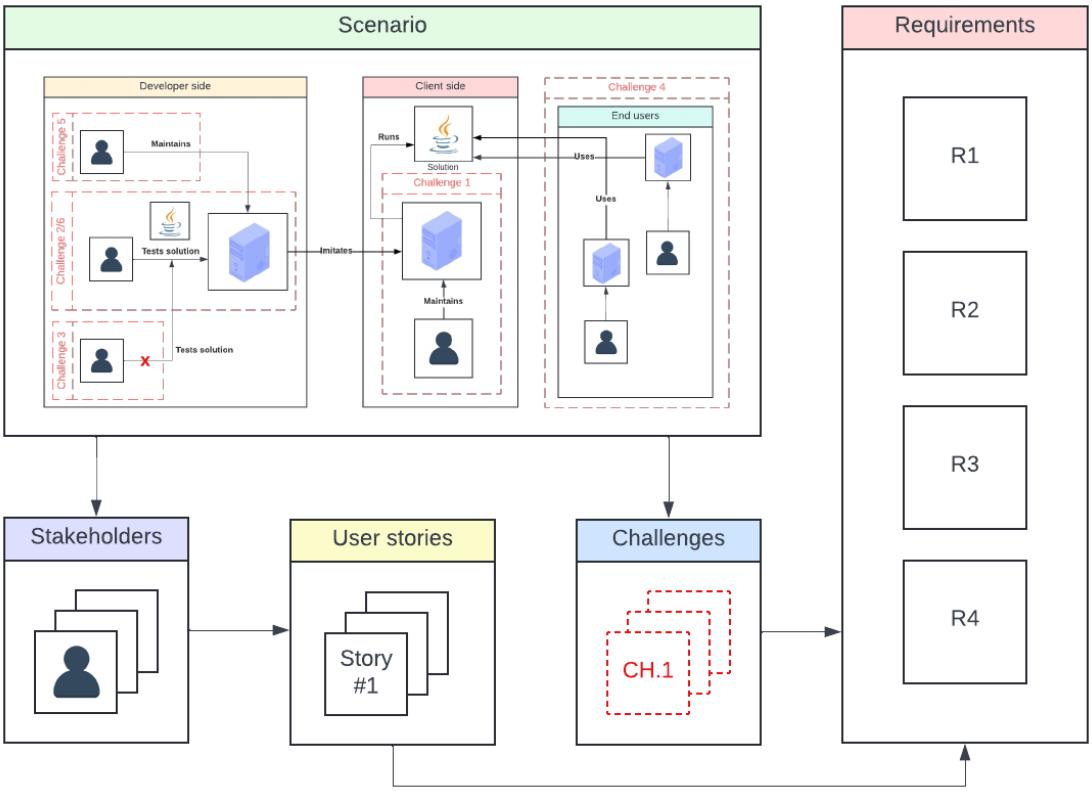

Figure 4.2: Requirements elicitation process

We have already defined a scenario in the introduction (Figure 1.1). This is the first step of the elicitation process, and once defined, we can derive the stakeholders. Moreover, from the perspective of the relevant stakeholders we define user stories that describe their goals (as a X, I want to Y, so that Z). By analyzing the scenario we can also outline challenges that we wish to address, which alongside the user stories will then be used to derive the requirements. Each user story results in a requirement, and should a challenge not be addressed we define a user story for it.

4.2 Pipeline requirements - R1

For the purposes of pipeline workflow design, the following requirements need to be fulfilled:

- **R1-1: Pipeline stimulus (Challenge 6)**:
  - **User story**: As a developer, I want the pipeline to begin running every time I introduce code changes, so that I don't have to manually run it.
  - **Requirement**: The pipeline needs to able to define a trigger event that will initiate pipeline execution. A pipeline should only be executed under certain conditions whose definition the pipeline needs to support.
- **R1-2: Pipeline segmentation**:
  - **User story**: As a DevOps engineer, I want to be able to divide the pipeline in to several stages, so that I can increase the readability and maintainability of pipeline code.
  - **Requirement**: The pipeline needs to be able to group together actions that aim to achieve a similar goal in to logical units. The pipeline also needs to support dependencies between each logical unit to ensure logical execution order.
- **R1-3: Pipeline autonomy (Challenge 6)**:
  - **User story**: As a developer, I want the pipeline to autonomously execute test, so that I don't spend time interacting with the pipeline.
  - **Requirement**: Once the pipeline is triggered, it needs to support the execution of the entire pipeline without any human intervention.
- **R1-4: Pipeline feedback**:
  - **User story**: As a developer, I want to be informed of the current state of the pipeline execution, so that I can follow the pipeline execution.
  - **Requirement**: The pipeline needs to support reporting of the current state of the pipeline execution back to the user.
- **R1-5: Pipeline concurrency**:
  - **User story**: As a DevOps engineer, I want to support multiple pipelines running in parallel, so that I can enable testing for various developers concurrently.
  - **Requirement**: The pipeline needs to be able to run in parallel with an arbitrary number of other pipelines, and should only be limited by hardware resources.

- **R1-6: Pipeline isolation (Challenge 3)**:
  - **User story**: As a developer, I want my pipeline to run in isolation from other pipelines, so that pipelines triggered by me are not affected by other pipelines running concurrently.
  - **Requirement**: The execution of the pipeline should not affect the execution of other pipeline running in parallel. Execution result of one pipeline must not influence the execution result of the other pipelines running concurrently.
- **R1-7: Pipeline idempotency (Challenge 2)**:
  - **User story**: As a DevOps engineer, I want to ensure no residue is left from a pipeline execution, to ensure consistency and idempotency between pipeline executions.
  - **Requirement**: Subsequent runs of the pipeline on the same code base should result in the same result.
- **R1-8: Pipeline termination**:
  - **User story**: As a DevOps engineer, I want the pipeline to stop or continue execution if a part of the pipeline fails, to ensure no resources are wasted or resources are properly cleaned up respectively.
  - **Requirement**: The pipeline needs to provide a mechanism for aborting the execution of the rest of the pipeline, should one part of the pipeline fail. The pipeline also needs to provide a mechanism for continuing the execution of the rest of the pipeline in explicitly defined cases when one part of the pipeline fails.
- **R1-9: Pipeline runtime environment specification**:
  - **User story**: As a DevOps engineer, I want to customize the pipeline execution environment, so that I can ensure proper dependencies are present and isolation is insured.
  - **Requirement**: The pipeline needs to support the ability to specify the environment that will be used for pipeline execution.
- **R1-10: Pipeline artifact persistence**:
  - **User story**: As a DevOps engineer, I want to extend the life-cycle of specified files beyond the pipeline stage or pipeline execution, so that these can be utilized in other stages or used for delivery/debugging respectively.
  - **Requirement**: The pipeline needs to support persistence of explicitly defined files between logical units of the pipeline, as well as explicit file persistence after pipeline execution.

4.3 Test structure requirements - R2

In order to insure that we can achieve all the needed cloud tests, the following list defines the requirements for the pipeline pertaining to cloud-tests:

- **R2-1: Test idempotency (Challenge 2)**:
  - **User story**: As a tester, I want the cloud test execution to not affect future execution results, so that I can ensure consistency and idempotency of cloud-tests.
  - **Requirement**: The pipeline needs to ensure that between consecutive cloud test no residue is left, meaning that running a same cloud test for the same code base results in the same test outcome. This also means that no state must be kept between the cloud tests, which means that the pipeline also needs to perform deployment and destruction of infrastructure for each test.
- **R2-2: Test concurrency (Challenge 3)**:
  - **User story**: As a tester, I want to run several cloud-tests concurrently, so that I can enable parallelization of cloud-tests.
  - **Requirement**: The pipeline needs to support running of arbitrary amount of cloud tests in parallel, and should only be limited by hardware resources.
- **R2-3: Test isolation (Challenge 3)**:
  - **User story**: As a tester, I want to ensure cloud-test isolation, so that cloud-test do not interfere with each other.
  - **Requirement**: The pipeline needs to ensure that each cloud test is performed in its own environment in order to ensure test isolation and prevent interference between tests. A cloud test's outcome must not influence the execution of other cloud tests.
- **R2-4: Test autonomy (Challenge 6)**:
  - **User story**: As a developer, I want the cloud-tests running in the pipeline to execute autonomously, so that I do not have to invest time interacting with cloud-tests.
  - **Requirement**: The pipeline needs to enable execution of cloud tests without requiring any human intervention, in order to ensure automation of the test execution.
- **R2-5: Test feedback**:
  - **User story**: As a developer, I want to be informed about the state of cloud-test execution, so that I can overseer cloud test outcome.
  - **Requirement**: The pipeline needs to be able to report the result of each cloud test to the user, regardless of the cloud test outcome.

4.4 (Cloud) Provisioning requirements - R3

In this section we will address technology specific requirements that the provisioning tool used needs to fulfill. All the following requirements solve the challenges 1, 4 & 5. These requirements also dictate the requirements for the cloud provider as the provisioning can only happen on resources that the cloud provider supports.

- **R3-1: Test environment definition**:
  - **User story**: As a tester, I want to be able to define resources through code in terms of network components, to be able to simulate the client environment.
  - **Requirement**: Specified through code, the provisioning tool needs to be able to provision the following resources:
    - **Network**
    - **Virtual machine**
    - **Router**
- **R3-2: Virtual machine hardware definition** :
  - **User story**: As a tester, I want to be able to define hardware resources of a virtual machine, so that I can more accurately simulate the client environment.
  - **Requirement**: Specified through code, the provisioning tool needs to be able to define the following resources:
    - **vCPU**
    - **RAM**
    - **Storage memory**
- **R3-3: Virtual machine software definition**:
  - **User story**: As a tester, I want to be able to define software configuration of the virtual machine in terms of the operating system running on the virtual machine, to more accurately simulate the client environment.
  - **Requirement**: Specified through code, the provisioning tool needs to be able to define the operating system that will run on top of this virtual machine.

4.5 Configuration management requirements - R4

In this section we will address technology specific requirements that the configuration management tool used needs to fulfill.

- **R4-1: Test step definition (Challenge 4)**:
  - **User story**: As a tester, I want to manipulate the state of the virtual machine through code, so that I can define its behavior in a cloud test.
  - **Requirement**: Specified through code, the configuration management tool needs to enable definition of steps that need to be performed in the test that are directly executed on the virtual machine.
- **R4-2: Test oracle (Challenge 6)**:
  - **User story**: As a tester, I want to define a passing or failing test through code, so that I do not have to manually check if a test passed or failed.
  - **Requirement**: The configuration management tool needs to be able to witness a test as passing or failing, where passing constitutes successful execution of all test steps, otherwise the test is considered as failing.
- **R4-3: Test autonomy (Challenge 6)**:
  - **User story**: As a tester, I want to define the cloud test workflow through code, so that I do not have to manually interact with the test execution.
  - **Requirement**: The configuration management tools needs to support execution of every test step without manual intervention of any kind.

CHAPTER 5

Expert interview and technology choice

5.1 Interview design

Before we begin with the implementation, we first have to decide what open-source technology stack to utilize to achieve the requirements described in Chapter 4. To answer this question we conducted interviews with various experts/practitioners at Hitachi Rail. The interviews were held in a 1-on-1 setting so as to not let the answers of other interviewees influence the others. All the interviews were recorded and performed in oral form, with the most noteworthy information summarized in the following section. The interview also follows a structured interview form [58] where the questions were predefined and followed a strict sequential order. Before each interview, we explained the topic to the interviewee, and what the goal of this work is, so as to not overburden each question with explanations and unnecessarily prolong the interview itself. The interviewees also agreed on being recorded for the scientific purposes of this work.

5.1.1 Interview structure

The interview is structured in to three logical sections that group together questions with a similar goal. The first section serves as the **introduction** that gathers relevant information about the interviewee that prove their competency in the matter. By the end of this section, we should be well versed with the background of the interviewee. The section consists of the two following questions:

- **Q.1: Please introduce yourself, what is your role in the company and how many years of experience do you have?**

- **Q.2: What is your experience with Infrastructure as Code, DevOps, and Cloud computing?**

In the second section, we aim to gather guidelines from the experts as to which **technology stack** they deem recommended and appropriate to achieve an automatic pipeline for the purposes of this work. By the end of this section, the interviewee should have recommended a technology stack as well as an alternative approach to offer guidance in case a different approach needs to be explored. The two following questions make up this section:

- **Q.3: What would you recommend as the open-source technology stack for the previously described pipeline?**
- **Q.4: What alternative approach/technology stack would you also recommend?**

The third, and also the last section, gathers insight from the interviewee about what **benefits** and **challenges/drawbacks** one might encounter by taking the approach of testing in the cloud. It also explores the types of **tests** the experts might deem noteworthy of exploration. It closes off with a very open question that leaves the freedom to the interviewee to add anything they want at the end. The four questions of this sections are:

- **Q.5: What benefits can one expect by taking this approach (testing in the cloud, test environment simulation in the cloud) compared to traditional testing methods?**
- **Q.6: What challenges would one encounter by taking this approach (testing in the cloud, test environment simulation in the cloud) compared to traditional testing methods?**
- **Q.7: What type of tests should/could one explore by testing in the cloud?**
- **Q.8: Would you like to add anything at the end?**

5.2 Interview results

5.2.1 Interviewee demographic

Five candidates where chosen and interviewed based on their experience, competence and expertise with the topic of IaC, CI/CD and Cloud computing. All relevant interviewee information is summarized in the following table 5.1. The experience column represents a non-exhaustive list of various different experiences the interviewee has had in the domain of IaC, CI/CD, and Cloud computing.

Int. num.	Role	Years of experience	Experience
1	Senior Software Engineer	40+	In-house development of automatic unit tests, Development of code quality tools, Development of CI strategies
2	Chief DevOps Engineer, Platform Architect, DevOps Engineer Expert	20+	Development of cloud native common stack, Assisting adoption of new technologies, Development of private cloud system using OpenStack
3	Senior System Engineer	20+ (6+ with cloud computing)	VMware infrastructure, Environment automation using Ansible and Openstack
4	Cloud Architect	18+ (6+ with OpenStack)	Project infrastructure development, High availability configuration, Containerization, Configuration/Maintenance/Deployment of OpenStack, Test environment automation
5	DevOps Engineer	4+	Networking architecture, Creation and setup of an air gapped cloud system, Bare metal provisioning

Table 5.1: The demographic of the interviewees/experts

5.2.2 Interview discussion

Let us first outline some major keypoints made by each expert. Summarized overview can be found in the Table 5.2 and Table 5.3, but some main talking points are presented as follows.

- **Interviewee/Expert #1**

  When comparing technologies, the expert emphasizes the API driven nature of OpenStack, while also recommending it as an optimal choice. They compare it to Proxmox, and consider it less capable for automation, but consider it a quick way of setting up infrastructure. Jenkins is mentioned as more flexible CI tool compared to Gitlab CI but they recommend Gitlab CI when starting from scratch.

Expert states that while Ansible, Puppet and Chef are all used in the company in some shape or form, they still rate Ansible as the better option for good readable configuration. For licensing reason OpenTofu is recommended. When asked for an alternative approach, they recommend using OpenTofu for both provisioning and configuration to avoid complexity. They also consider no alternative to Ansible. In case something else besides OpenTofu wants to be used, Pulumi is recommended. The main benefit they point out is the automation of the whole process. They also point out how a big challenge in this approach is the complexity of the setup which usually requires a specialist in the team. When giving examples of appropriate tests, they recommend tests with arbitrary networks, smaller and bigger ones, and also showing how unit tests can be done upfront and integration tests in the cloud.

- **Interviewee/Expert #2**

  When discussing technologies, the expert suggested OpenTofu as the easier approach compared to Pulumi, explaining how they have big projects in the company using Terraform/OpenTofu. They state how Kubernetes might be a good approach in case of much bigger environments but introduces more complexity. In case of greenfield deployment Gitlab is given as a starting point for CI/CD purposes. Ansible is given as an example of how provisioning is done in the company, where virtual machines are run on top of OpenStack. Terraform/OpenTofu is praised as a good tandem with OpenStack. The expert also suggests Crossplane in case of Kubernetes as it is really tied to it. As an alternative approach, the experts suggests reducing the complexity by removing layers from the technology stack, with one example being the removal of the virtualization layer. Automation is highlighted as the major benefit of this approach, and the immediate feedback of automatic cloud tests. Again, the expert brings up complexity and non-triviality as one drawback requiring a lot of knowledge for both development and debugging. When asked about the types of test to be explored, they highlight functional tests, performance tests, and resilience testing, giving chaos engineering as an example.

- **Interviewee/Expert #3**

  In the beginning of the interview, the expert brought to attention that his experience is mostly with licensed software. Despite this they gave input regarding some open-source technologies, but abstained from answering in some cases like the alternative technology stack. They emphasized that their recent experience was with deploying fully automated environments with Ansible and OpenStack. When discussing IaC they emphasize how they've used Puppet quite a lot but would still recommend Ansible, stating that they prefer the way Ansible works over Puppet. The expert also points out that although he had previously used Terraform, their job is usually done by the time Terraform can even run. When discussing benefits and challenges, they stress how automation removes errors when deployment is done by hand, but they prefer to use licensed tools because of easier troubleshooting and debugging.

- **Interviewee/Expert #4**

  When asked about the optimal technology stack, the expert states that they use Terraform, and despite its ability to also do configuration management they prefer using it in tandem with Ansible. They explain how Ansible takes over immediately after infrastructure creation by Terraform. As a drop-in replacement for Terraform, OpenTofu is suggested. Notably, Jenkins is praised as more powerful compared to Gitlab CI, while introducing more complexity. Despite this they view Gitlab as the better solution, stating that they haven't encountered a problem that they couldn't solve with Gitlab. They emphasize the versatility of OpenStack for its ability of choosing what type of services one needs (e.g. DNS as a service). When comparing the learning curve, they admit that Proxmox does have a less steep learning curve compared to OpenStack, but that it also comes with less capabilities. They state how the use case plays an important role when deciding between the two, and that for a smaller company OpenStack may prove to be superfluous. Puppet was brought up as having a steep learning curve, where Ansible comes as a better option due to its readability and better understanding of what is actually happening in the background, as it simply executes tasks from beginning to the end. When arguing for the benefits of testing in the cloud, the expert brings up an example of a static environment used for testing in the company that became hard to maintain due to its age, and lack of reproducibility, highlighting automation as the key solution in case of virtual environments. The last example given was a testing scenario that would be possible to achieve in the cloud where the solution the expert has worked on needs to be tested for over 150 clients, which could be achieved with virtual machines for each of them.

- **Interviewee/Expert #5**

  During the technology stack discussion, Gitlab was praised for providing many different services all in one place, like offering registry storage, code storage and CI/CD capabilities at the same time. Grafana was also suggested as a monitoring tool but is noted that they don't deem it mandatory. Although they outline OpenTofu/Terraform and Ansible as the choice for the technology stack, the expert emphasizes the similarity between a lot of technologies mostly using the same approach but different names and implementations, so their recommendations most likely stem from experience of using those technologies like OpenStack. They mention how Terraform/OpenTofu and Ansible can fill in for the other and both can provision and manage configuration as something worth considering when taking an alternative approach. When arguing for the benefits they bring up how in the past testing was usually done on a server that was physically maintained and required physical presence for configuration. Everything had to be manually connected and configured. They emphasize the ability to simulate a real physical network with actual hops, instead of just testing using loop back network interface to simulate communication.

Int. num.	Q.3 (Optimal tech. stack)	Q.4 (Alternative stack)
1	Openstack Ansible OpenTofu Gitlab CI	Proxmox OpenTofu/Pulumi Jenkins
2	OpenStack OpenTofu Kubernetes (depending on use case) Ansible Gitlab CI	(Reduce complexity) Kuberenetes Crossplane
3	OpenStack Kayobe (Kolla Ansible) Ansible Gitlab with Jenkins	(no alternative recommended)
4	OpenStack Ansible OpenTofu Gitlab CI	Proxmox Ansible OpenTofu Jenkins
5	OpenStack Ansible OpenTofu Gitlab CI	Ansible only or OpenTofu only Proxmox Jenkins

Table 5.2: Tabular overview of the interviewee/expert answers to Q.3, and Q.4

Int. num.	Q.5 (Benefits)	Q.6 (Challenges)	Q.7 (Tests)
1	Automatic rollout, Parallelized testing, No need to buy services, Network emulation	Mismatch between virtual and client environment, Concrete hardware simulation, Complex setup	Arbitrary configuration (arbitrary networks), Integration tests
2	End-to-end automation, Testing at any point in time, Immediate feedback on changes	Solution complexity, Non-triviality	Functional tests, Performance tests, Resilience tests
3	No need for manual maintenance, Scalability, reusability, flexibility Automation	Open-source being developed by a lot of different people (inconsistency), Different applications from different people	Reliability tests, Ensuring proper cleanup between tests
4	Reproducibility of the tests/test environment, Ability to scale up an down	System complexity, Knowledge of underlying infrastructure	Rollout testing, Performance/Stress tests, Scalability tests
5	No need for manual maintenance, Every user has their own environment , Environment definition through code	Steep learning curve, Synergizing different technologies	Validation of network communication, Performance/Stress tests, Chaos engineering

Table 5.3: Tabular overview of the interviewee/expert answers to Q.5, Q.6, and Q.7

5.3 Technology choice

The goal of the interviews was to utilize the experience and expertise of the interviewees to draw conclusions on what the recommended technology stack would be. By analyzing the answers, we can see a clear preference towards the same technologies, showing unanimous support towards the following technology stack as the recommended one:

Objective	Technology
Cloud computing	OpenStack
Provisioning	OpenTofu
Configuration management	Ansible
Continuous Integration (pipeline)	Gitlab CI

Table 5.4: Recommended technology stack

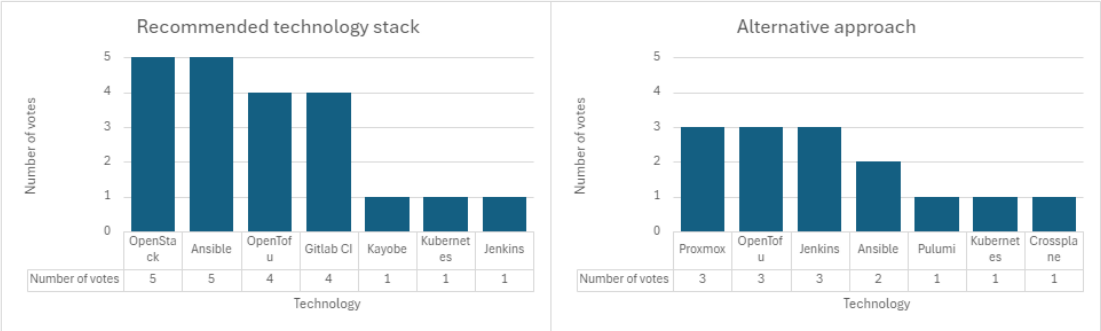

Figure 5.1: Bar chart showing vote distribution across technologies mentioned by the experts (non-technology-specific recommendations are not shown)

Our goal is to also recommend an alternative approach, and by analyzing the answers of the experts we can see mixed answers as to what constitutes as an alternative to the main stack (5.5). Still, looking at Figure 5.1 we can see that the majority agrees on the following:

Objective	Technology
Cloud computing	Proxmox
Provisioning	OpenTofu
Configuration management	Ansible
Continuous Integration (pipeline)	Jenkins

Table 5.5: Alternative technology stack

CHAPTER 6

Architecture and implementation

6.1 Architecture

6.1.1 Problem description

Let us consider a practical use case scenario. A simple endpoint written in Python that serves GET requests for adding two numbers together. The goal is to perform an integration test in the cloud, where we will simulate a user sending a request over the network and expecting a correct result.

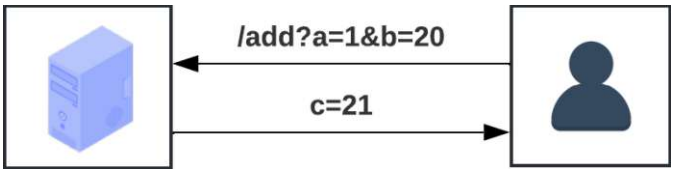

Figure 6.1: Interaction of the endpoint

6.1.2 Pipeline description

The pipeline will be divided into the following stages:

- **Build**: The objective of this stage is to build the solution from the source code. However, should it fail doing so, it stops the rest of the pipeline.
- **Unit-test**: The goal of this stage is to perform unit tests that will validate the correctness of the implementation, in order to prevent unnecessary running of cloud-tests.
- **Cloud-test**: This stage performs the actual cloud testing and is divided in to three jobs:

  - **Cloud-deploy**: In this job we will start provisioning our cloud infrastructure by deploying our virtual environment to the cloud.
  - **Test**: The goal of this job is to perform the actual tests once the deployment of the environment is done
  - **Cloud-cleanup**: The objective of this job is to destroy and cleanup the previously deployed cloud infrastructure while ensuring no residue
- **Deploy**: This stage is tasked with deploying the actual solution in case all other tests pass.

6.1.3 Cloud test description

We will describe the required test environment configuration that we wish to virtualize and deploy to our cloud. The environment requires the following components:

- **VM 1**: Server
- **VM 2**: Client
- **Network 1**: Network containing VM1
- **Network 2**: Network containing VM2
- **Router**: Connecting the two networks

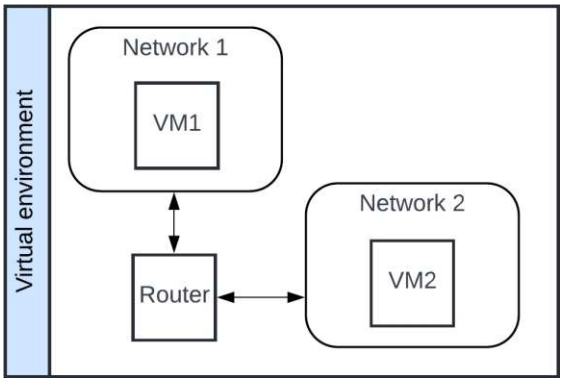

Figure 6.2: Cloud test environment configuration

Now we can describe the workflow of the test with the following steps:

- **Step 1**: Server begins serving the addition endpoint
- **Step 2**: Client sends a GET request to the server
- **Step 3**: Client validates a successful connection
- **Step 4**: [If step 3 succeeds] Client validates the correctness of the result

A successfully passing test requires both a successful connection and a correct result.

6.1.4 Architecture

Let us take an overview of the pipeline architecture. Using the technology stack derived in 5.3 we can derive the following diagram (Figure 6.3) of technology interaction. The diagram also describes an example workflow of the pipeline divided in to several stages.

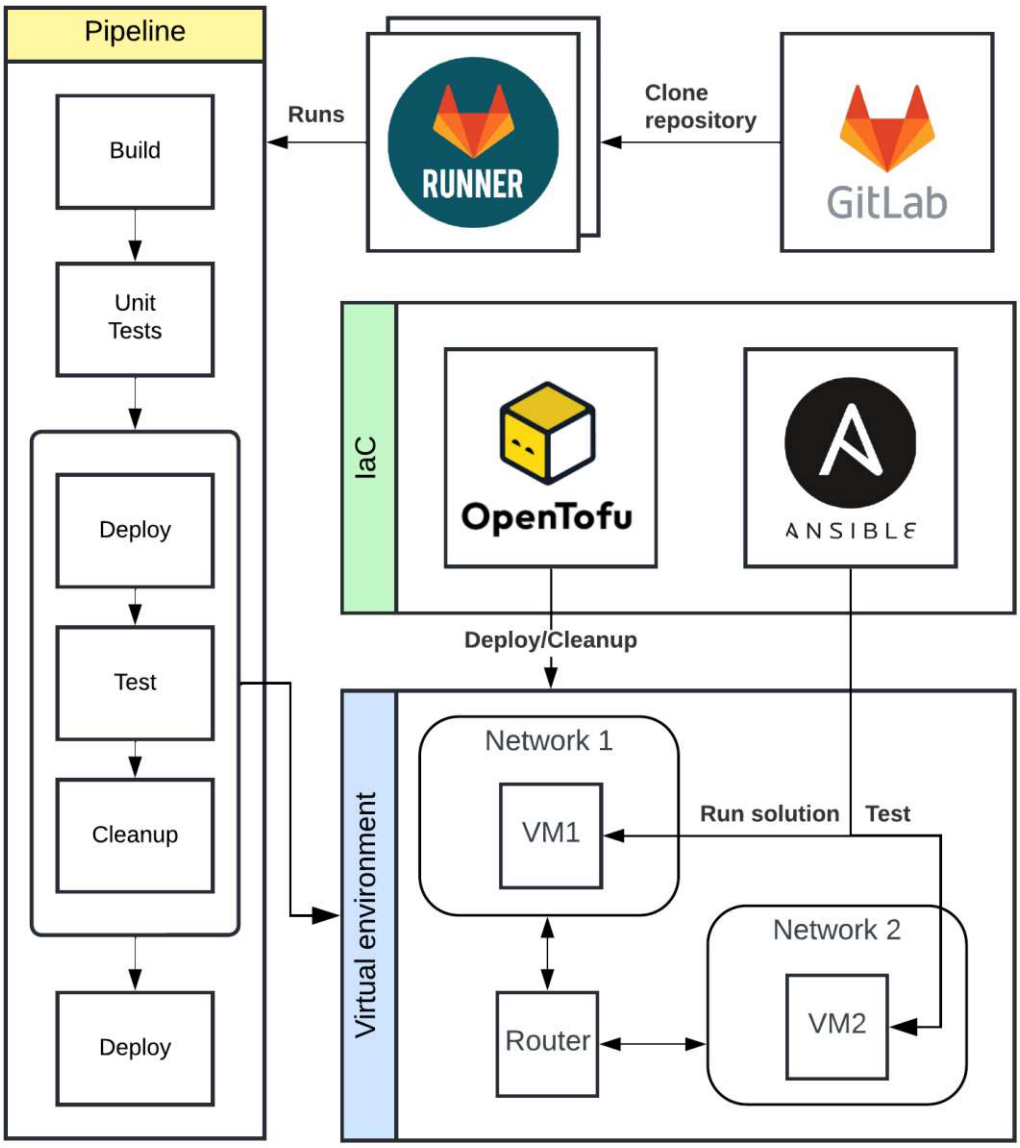

Figure 6.3: Pipeline infrastructure diagram

6.2 Setup and configuration

Before we begin with the actual implementation we need to perform the necessary setup steps. We assume the worst case scenario, that the entire infrastructure will be self-hosted on premise and as such we have to demonstrate all the needed configuration steps one would need to perform which serve as the prerequisite of the implementation. We begin by setting up our Gitlab and Gitlab Runners that will run the actual pipeline. Due to the scope of this work, OpenStack setup is not covered.

6.2.1 Gitlab

As we will be hosting our own Gitlab we need to cover the setup of Gitlab itself before we can use Gitlab CI. Gitlab supports various ways of installation which offers us flexibility when choosing which approach to take [10]. We may install it as a Linux package, deploy it over Kubernetes, or run it within a Docker container. To ensure maximum isolation from the rest of the components, we will use OpenStack Horizon to manually spawn an Openstack Compute instance, and run Gitlab within a Docker container. This in essence means that will have our own virtual machine with the sole purpose of hosting Gitlab. According to the minimum requirements for Gitlab, the underlying machine needs to have at least 4GBs of RAM. To ensure this, we spawn an instance with the following flavor:

Resource	Amount
VCPUs	2
RAM	8GB
Size	40GB

Table 6.1: Gitlab instance flavor

```
sudo -E docker run --detach \
    --hostname gitlab.masterthesis.cloud.at.gts  \
    --env GITLAB_OMNIBUS_CONFIG="external_url 'http://gitlab.
       masterthesis.cloud.at.gts'" \
    --publish 443:443 \
    --publish 80:80 \
    --publish 22:22   \
    --name gitlab   \
    --restart always   \
    --volume /home/ubuntu/gitlab/config:/etc/gitlab   \
    --volume /home/ubuntu/gitlab/logs:/var/log/gitlab   \
    --volume /home/ubuntu/gitlab/data:/var/opt/gitlab   \
    --shm-size 256m   gitlab/gitlab-ce:latest
```

Listing 6.1: Docker command to run GitLab

It should be noted that the instance upon which Gitlab is running, needs to be added to a security group that allows ingress for ports 22 (SSH), 80 (HTTP), 443 (HTTPS). Since we are also going to customize the runtime environment of the pipeline, we will build our own Docker image and upload it to our own registry. Note: for simplicity we are going to use HTTP for our internal cloud, as HTTPS would require additionally certificates.

```
gitlab_rails['registry_enabled'] = true
gitlab_rails['registry_host'] = "gitlab.masterthesis.cloud.at.gts"
gitlab_rails['registry_port'] = "443"
registry['enable'] = true
registry['registry_http_addr'] = "0.0.0.0:443"
```

Listing 6.2: Gitlab registry configuration

6.2.2 Gitlab Runner

As is the case with Gitlab, the actual pipeline runner (Gitlab Runner), can also be installed in various ways [18], but for consistency we will also install it as a Docker container, running on a separate virtual machine. We will use the same flavor as the Gitlab virtual machine. To run the Gitlab Runner as a container:

```
docker run -d --name gitlab-runner --restart always   -v /srv/gitlab-
   runner/config:/etc/gitlab-runner   -v /var/run/docker.sock:/var/
   run/docker.sock   gitlab/gitlab-runner:latest
```

Listing 6.3: Docker command to run GitLab Runner

For the runner to be able to execute pipeline jobs, we need to register it to Gitlab:

```
docker run --rm -it -v /srv/gitlab-runner/config:/etc/gitlab-runner
   gitlab/gitlab-runner register
```

Listing 6.4: Docker command to register the runner

The registration process is interactive but the runner can still be configured manually:

```
[[runners]]
  name = "gitlab-runner-new"
  url = "http://gitlab.masterthesis.cloud.at.gts"
  id = 4
  token = "aaa-bbb-ccc"
  token_obtained_at = 2024-09-23T11:46:26Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "docker"
  [runners.custom_build_dir]
  [runners.cache]
    MaxUploadedArchiveSize = 0
    [runners.cache.s3]
```

```
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    tls_verify = false
    image = "ubuntu:latest"
    privileged = true
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.sock"]
    shm_size = 0
    network_mtu = 0
```

Listing 6.5: Runner configuration

We are going to use Docker as the executor for the pipeline, with the default image being Ubuntu. We will overwrite this in the pipeline and select our own image for the execution of the pipeline. Furthermore we will build that image (Appendix A.4), and for that we need to enable privileged access.

6.2.3 Sample program implementation

As described in problem description (Subsection 6.1.1) we define a use case that will demonstrate the capabilities of the pipeline, where we have a typical case of communication between a server and a client. We can achieve this in Python in the following way:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/add', methods=['GET'])
def add_numbers():
    return jsonify({"result": (request.args.get('num1', type=float)+
        request.args.get('num2', type=float))})

if __name__ == '__main__':
    app.run(debug=True)
```

Listing 6.6: Sample program implementation

To make the Unit-test stage meaningful we also define a simple unit test:

```
import unittest
import json
from addition_endpoint import app

class AddTwoNumbersTestCase(unittest.TestCase):
    def testGet(self):
        self.endpoint = app.test_client()
```

```
        self.endpoint.testing = True
        response = self.endpoint.get('/add', query_string={'num1': '1
            ', 'num2': '20'})
        self.assertEqual(response.status_code,200)
        self.assertEqual(json.loads(response.data)['result'],21.0)

if __name__ == '__main__':
    unittest.main()
```

Listing 6.7: Sample program test implementation

6.3 Pipeline implementation

Now we will take a look at how we implemented the actual pipeline that will serve as the backbone for both OpenTofu and Ansible. The goal is to divide the pipeline into stages. We will implement the pipeline stage by stage and as such we will begin with the first stage two stages.

```
stages:
  - build
  - unit-test
  - cloud-test
  - deploy
```

Listing 6.8: Pipeline stages

6.3.1 Build and Unit-test

As these two stages are short we will cover both of them in this subsection. Build stage usually requires building and compiling the solution, as Python doesn't require compiling we will just package it using Python's package builder [27]. Unit-test stage will run a simple unit test designed to test if the enpdoint produces the correct result.

```
build:
  stage: build
  before_script:
    - cd poc/python
  script:
    - python3 -m build

unit-test:
  stage: unit-test
  before_script:
    - cd poc/python
  script: python3 -m unittest discover -s tests
```

Listing 6.9: Build nad Unit-test stages

6.3.2 Cloud-test

To make the pipeline code more readable we will divide cloud-test stage in to multiple jobs. The first job, cloud-deploy, will be tasked with deploying the actual infrastracture and for that we will use OpenTofu. Let us define main resources that we will need to provision. We will start with the top down approach, and define the two networks needed.

```
1
2  resource "openstack_networking_network_v2" "server_network" {
3    name = "server_network"
4  }
5
6  resource "openstack_networking_network_v2" "client_network" {
7    name = "client_network"
8  }
9
10 resource "openstack_networking_subnet_v2" "server_subnet" {
11   name       = "server_subnet"
12   network_id = openstack_networking_network_v2.server_network.id
13   cidr       = "21.21.0.0/24"
14   ip_version = 4
15 }
16
17 resource "openstack_networking_subnet_v2" "client_subnet" {
18   name       = "client_subnet"
19   network_id = openstack_networking_network_v2.client_network.id
20   cidr       = "56.56.0.0/24"
21   ip_version = 4
22 }
```

Listing 6.10: Network and subnet creation

Here we've chosen arbitrary values for the subnet and their values do not play an important role. Following the definition of the networks we need to provision a router connecting these two networks. For that we will define a router and add two interfaces to it, each connected to a network.

```
1  resource "openstack_networking_router_v2" "router" {
2    name                = "router"
3    external_network_id = data.openstack_networking_network_v2.
       lab_teddy.id
4  }
5
6  resource "openstack_networking_router_interface_v2" "
     router_interface_server_network" {
7    router_id = openstack_networking_router_v2.router.id
8    subnet_id = openstack_networking_subnet_v2.server_subnet.id
9  }
10
```

```
resource "openstack_networking_router_interface_v2" "
    router_interface_client_network" {
  router_id = openstack_networking_router_v2.router.id
  subnet_id = openstack_networking_subnet_v2.client_subnet.id
}
```

Listing 6.11: Router and interface

Now we are ready to spawn the virtual machines that will serve the roles of the server and the client.

```
resource "openstack_compute_instance_v2" "server" {
  name            = "server"
  image_name      = "Ubuntu 24.04 LTS AT.GTS Docker 2024-08-05"
  flavor_name     = "general-1C-1G-5GB"
  key_pair        = "id_rsa"
  security_groups = ["all-icmp", "rdp-ubuntu", "ssh+https", "flask"]

  network {
    uuid = openstack_networking_network_v2.network1.id
  }
}

resource "openstack_compute_instance_v2" "client" {
  name            = "client"
  image_name      = "Ubuntu 24.04 LTS AT.GTS Docker 2024-08-05"
  flavor_name     = "general-1C-1G-5GB"
  key_pair        = "id_rsa"
  security_groups = ["all-icmp", "rdp-ubuntu", "ssh+https"]

  network {
    uuid = openstack_networking_network_v2.network2.id
  }
}
```

Listing 6.12: Instance creation

The last step is exposing these virtual machines using the concept of floating IPs in order for Ansible to be able to access them and perform the tests. Floating IPs are taken from the network to which Ansible has access to (external network), in our case this is a network named outside_network.

```
data "openstack_networking_network_v2" "outside_network" {
  name = "outside_network"
}

resource "openstack_networking_floatingip_v2" "server_floating_ip" {
  pool = "lab_teddy"
}
```

```
8
9  resource "openstack_networking_floatingip_v2" "client_floating_ip" {
10   pool = "lab_teddy"
11 }
12
13 resource "openstack_compute_floatingip_associate_v2" "
       fip_server_association" {
14   floating_ip = openstack_networking_floatingip_v2.server_floating_ip
         .address
15   instance_id = openstack_compute_instance_v2.server.id
16 }
17
18 resource "openstack_compute_floatingip_associate_v2" "
       fip_client_association" {
19   floating_ip = openstack_networking_floatingip_v2.client_floating_ip
20   .address
21   instance_id = openstack_compute_instance_v2.client.id
22 }
```

Listing 6.13: Floating IP definition and association to instances

Now we can define the actual job and how it deploys the cloud infrastructure as above described through code.

```
1  cloud-deploy:
2    stage: cloud-test
3    variables:
4      TF_CLI_CONFIG_FILE: "${CI_PROJECT_DIR}/poc/opentofu/.tofurc"
5    before_script:
6      - echo "$CLOUDS_YAML_64" | base64 -d > poc/opentofu/clouds.yaml
7      - cd poc/opentofu
8    script:
9      - tofu init
10     - tofu plan
11     - tofu apply -auto-approve
12   artifacts:
13     paths:
14       - poc/opentofu/terraform.tfstate
15       - poc/ansible/inventory
```

Listing 6.14: Pipeline cloud-test stage, cloud-deploy job

Before we begin deploying we have to setup `cloud.yaml` (Appendix A.1) in order to enable OpenTofu to connect to OpenStack. Followed by this, we enter the necessary directory and initalize OpenTofu. We can preview what changes will take place after deploying the infrastructure which can be done with `tofu plan`. This serves as additional information to the reader of the pipeline who observes its execution. We apply the plan and the deployment begins. As OpenTofu keeps information about the infrastructure which will be needed in the cleanup job, we need to persist this file which we do using

`artifacts` keyword. We also persist the inventory file that will be explained in the following section.

Bridging OpenTofu and Ansible

Once OpenTofu is done deploying the infrastructure, the next stage of the cloud-test begins, which is the actual test that is going to be performed. Actual execution of the test is performed by Ansible but in order to connect to the virtual machines, Ansible needs to know their respective floating IPs. As explained in Section 2.3.4 Ansible keeps track of all hosts in its inventory, so for this purpose we will utilize `template_file` provider and using a template file manually create the inventory file needed for Ansible. OpenTofu also supports Ansible as a provider but since we do not want to run Ansible immediately, but rather extract it into its own job, this approach won't be considered. In OpenTofu we have to define various outputs that will be inserted into the template, we begin by extracting the most important data:

```
output "server_internal_ip" {
  value = openstack_compute_instance_v2.server.access_ip_v4
}

output "client_internal_ip" {
  value = openstack_compute_instance_v2.client.access_ip_v4
}

output "server_floating_ip" {
  value = openstack_networking_floatingip_v2.server_floating_ip.
      address
}

output "client_floating_ip" {
  value = openstack_networking_floatingip_v2.client_floating_ip.
      address
}

data "template_file" "ansible_inventory" {
  template = file("../ansible/inventory.tpl")
  vars = {
    server_floating_ip = openstack_networking_floatingip_v2.
        server_floating_ip.address
    server_internal_ip = openstack_compute_instance_v2.server.
        access_ip_v4
    client_floating_ip = openstack_networking_floatingip_v2.
        client_floating_ip.address
    client_internal_ip = openstack_compute_instance_v2.client.
        access_ip_v4
  }
}
```

Listing 6.15: Exporting IP addresses to Ansible inventory using a template file

Now we define the template file that will be instantiated after deployment:

```
[network1]
server ansible_host=${server_floating_ip} internal_ip=${
    server_internal_ip} ansible_user=ubuntu ansible_private_key_file
    =~/.ssh/id_rsa/id_rsa ansible_ssh_common_args='-o␣
    StrictHostKeyChecking=no'

[network2]
client ansible_host=${client_floating_ip} internal_ip=${
    client_internal_ip} ansible_user=ubuntu ansible_private_key_file
    =~/.ssh/id_rsa/id_rsa ansible_ssh_common_args='-o␣
    StrictHostKeyChecking=no'
```

Listing 6.16: Inventory template

Once deployment to the cloud is done we begin the execution of the test. In Ansible we can define tasks to be executed in a playbook. We can write a playbook that will define step by step how the test is to be executed. We begin by considering the server side, and make sure that virtual machine is booted up and necessary dependencies are installed.

```
- name: Setup server
  hosts: server
  gather_facts: no
  become: true
  tasks:
    - name: ssh wait
      wait_for:
        host: "{{␣hostvars['first_vm'].ansible_host␣}}"
        port: 22
        timeout: 300
        state: started
      delegate_to: localhost

    - name: Ensure Python is installed
      apt:
        name: python3
        state: present
      retries: 10
      delay: 5

    - name: Ensure pip is installed
      apt:
        name: python3-pip
        state: present
      retries: 20
```

```
      delay: 3

    - name: Ensure flask is installed
      apt:
        name: python3-flask
        state: present
```

Listing 6.17: Ansible task for server setup

Now we copy the solution to the server machine and run it. This will conclude the responsibilities of the server side.

```
- name: Copy Python Flask application to the target machine
  copy:
    src: ../python/addition_endpoint.py
    dest: ./addition_endpoint.py
    mode: '0755'

- name: Run python program
  shell: "nohup␣flask␣--app␣addition_endpoint␣run␣--host=0.0.0.0␣>␣/
     var/log/flask.log␣2>&1␣&"
```

Listing 6.18: Ansible server setup: solution setup

Now we consider the client side, we wait for the server side to start listening and serving on port 5000 (Flask port), followed by sending a request and validating it.

```
- name: Setup client and send a request to the server
  hosts: client
  gather_facts: no
  tasks:
    - name: Wait for Flask app to be up and running on first_vam
      wait_for:
        host: "{{␣hostvars['first_vm'].internal_ip␣}}"
        port: 5000
        delay: 5
        timeout: 60

    - name: Send GET request to /add on the Flask app
      uri:
        url: "http://{{␣hostvars['first_vm'].internal_ip␣}}:5000/add?
           num1=1&num2=20"
        method: GET
      register: response

    - name: Fail if response isnt OK (200)
      fail:
        msg: "Server␣responded␣with␣status␣{{␣result.status␣}}"
      when: response.status != 200

```

```
23      - name: Fail if the result is not correct
24        fail:
25          msg: "Server␣responded␣with␣result␣=␣{␣response.json.result␣
              }}"
26        when: response.json.result != 21.0
27
28      - name: Display the response from the Server app
29        debug:
30          msg: "The␣result␣of␣adding␣num1=1␣and␣num2=20␣is␣{␣response.
              json.result␣}}"
```

Listing 6.19: Ansible task for client setup and testing

Now we define the job in the pipeline. We need to make the job dependent on deployment as testing can only take place on a successful deployment task. For that we use the `needs` keyword. We print the contents of the inventory as additional debug information. As Ansible connects using SSH we also have to setup a private and a public key for this as done in the `before_script` part.

```
1  test:
2    stage: cloud-test
3    before_script:
4      - mkdir -p ~/.ssh/id_rsa/
5      - chmod 700 ~/.ssh
6      - echo "$SSH_PUBLIC" | base64 -d > ~/.ssh/id_rsa/id_rsa.pub
7      - echo "$SSH_PRIVATE" | base64 -d > ~/.ssh/id_rsa/id_rsa
8      - chmod 600 ~/.ssh/id_rsa/id_rsa
9      - cd poc/ansible
10   script:
11     - ansible-inventory -i inventory --list
12     - ansible-playbook -v -i inventory ping.yml
13   needs:
14     - cloud-deploy
```

Listing 6.20: Pipeline cloud-test stage, test job

To conclude the cloud-test stage we are only missing the cleanup part. The biggest obstacle to solve here is persisting the `tfstate` file from the deploy job which we have already done, and ensuring that this job is performed regardless of the result of test itself. This can be done by specifying a condition in `when` to `always`. OpenTofu destroy uses `tfstate` to destroy everything deployed in the deploy job.

```
1  cloud-cleanup:
2    stage: cloud-test
3    variables:
4      TF_CLI_CONFIG_FILE: "${CI_PROJECT_DIR}/poc/terraform/.tofurc"
5    before_script:
6      - echo "$CLOUDS_YAML_64" | base64 -d > poc/terraform/clouds.yaml
7      - cd poc/terraform
```

```
8   script:
9     - tofu init
10    - tofu destroy -auto-approve
11  needs:
12    - [cloud-deploy, test]
13  when: always
```

Listing 6.21: Pipeline cloud-test stage, cloud-cleanup job

The last stage, deployment, is covered in detail in Appendix A.3. Gitlab offers a graphical user interface for interacting with the pipeline in order to see exact results of each job and stage. A successful pipeline execution in our case would like Figure 6.4.

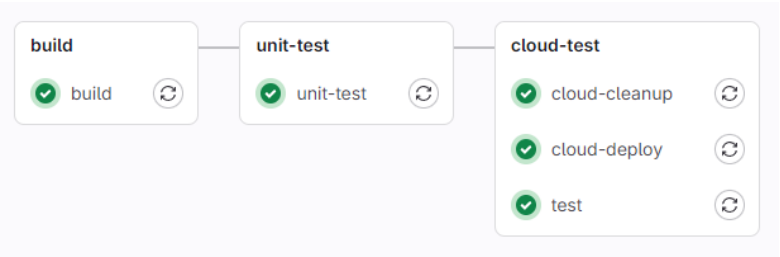

Figure 6.4: Gitlab GUI presenting the result of the pipeline execution

CHAPTER 7

Evaluation

7.1 Hitachi Rail use case

At Hitachi Rail, the innovation team is developing a moving block system (MBS) for railway signaling. In a fixed block system, the train tracks are divided into fixed segments that can be occupied by a train. Information about block occupancy is transmitted through visual signals that indicate whether or not the train behind in previous blocks should proceed or stop. In an MBS these blocks are dynamically calculated in real time (following the train) and they ensure that enough track space is left in front and behind the train. In case of the innovation team, the MBS consists of various components, while the most important ones, and the ones relevant to our use case are the following:

- Advanced Protection System (APS)
- Plan execution (PE)

Another component relevant to us for testing is the simulator that simulates actual trains, and track objects. This simulator is used for testing the functionalities of the MBS but is not considered a deliverable (is not delivered to the client). The innovation team has some tests that are automated, like unit tests and acceptance tests, but currently the solution is tested manually by running the above mentioned three components and performing actions through a simulator. The simulator allows for a web UI that visually depicts tracks and trains.

Our goal is to take the current testing approach of manual testing where test outcome is determined by visual inspection, and automate it by deploying the needed infrastructure to the cloud and performing the tests, automated through a use of a pipeline.

7.1.1 Solution approach

In the current setup there exists a solution repository that contains the code and a simple pipeline that builds the code, and runs the automated unit and acceptance tests in the process. In order not to overburden the solution repository we will create a testing repository with its own pipeline that will be triggered by the solution repository pipeline and will report back the result to it. That way, the users of the solution repository pipeline will see that there is a new cloud test stage but it will only be a trigger for the other testing pipeline. In Gitlab CI this concept is called multi-project pipeline [11]. Since we will be testing the distributed deployment of the components, we will create a virtual machine for each, all contained within one network. In order to deploy the solution to each virtual machine we will containerize each component. For simplicity, we will only cover building, containerization and the actual cloud testing, with the focus on the test, as the infrastructure follows a very similar approach to the prototype implementation.

7.1.2 Workflow

Let us describe the workflow of the cloud testing process for our use case. We will cover the workflow step by step. Figure 7.1 illustrates the workflow.

- **Step #1:** Each component is built.
- **Step #2:** Each component is containerized and pushed to the Gitlab container registry
- **Step #3:** Cloud tests begin
- **Step #4:** The cloud testing pipeline is triggered by the solution pipeline and begins execution
- **Step #5:** Deploy stage uses OpenTofu to deploy the infrastructure to the cloud
- **Step #6:** Cloud testing begins
  - **Step #6.1:** Ansible pulls the container images from the registry of the solution repository
  - **Step #6.2:** Ansible installs the solution on each three machines
  - **Step #6.3:** Ansible begins testing by sending various commands to the simulator
- **Step #7:** Cleanup job begins and using OpenTofu destroys deployed infrastructure
- **Step #8:** Testing pipeline finishes and reports back to the Cloud tests job of the solution repository ending the workflow

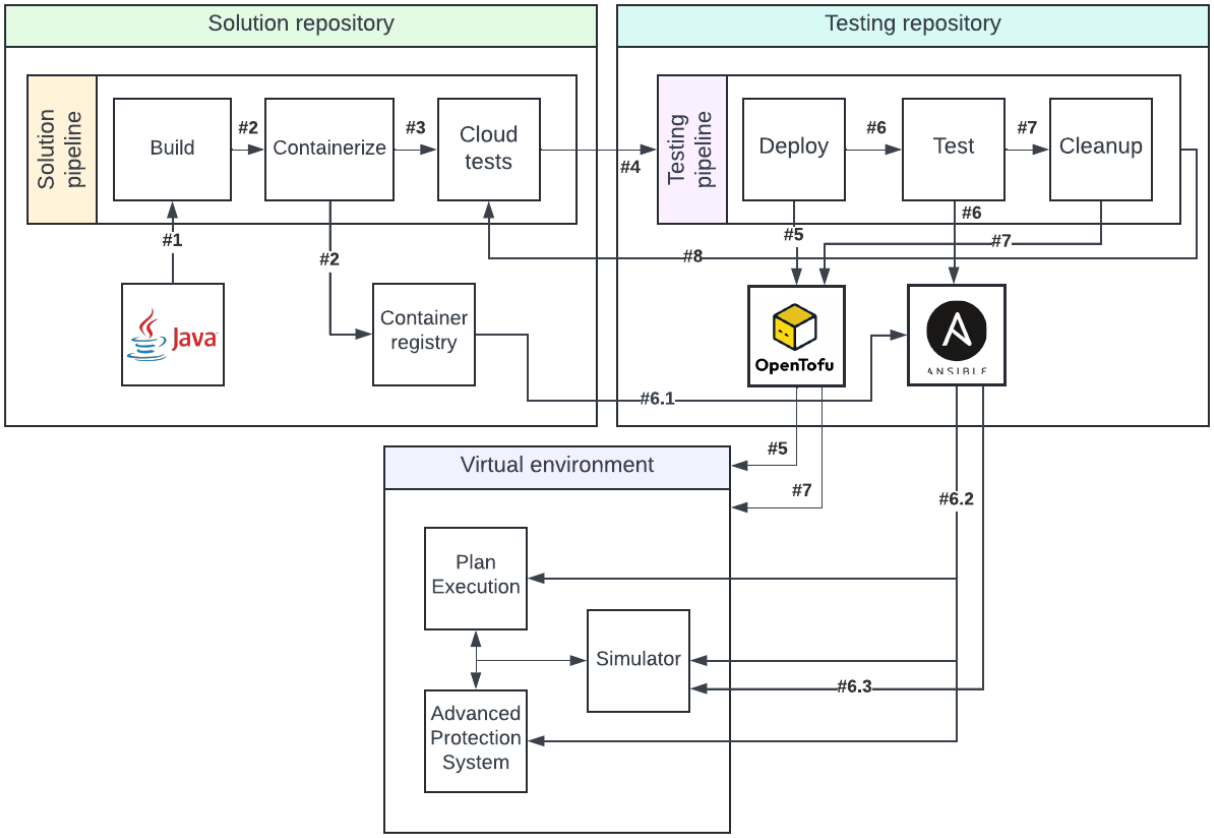

Figure 7.1: Hitachi Rail use case workflow diagram

7.1.3 Building and containerization

In order to test the interaction of all three components, each has to be containerized using Docker along with their respective dependencies in the form of configurations. For this purpose we define a Dockerfile as a recipe to building the actual image, and a script that runs the solution which will serve as the entry point to the solution. We now present how the simulator component is containerized. For simplicity we only cover the simulator as the rest of the components follow the same approach.

```
1  FROM gitlab:5005/tat/maxd/ubuntu-graal-docker:jdk21
2
3  EXPOSE 8080
4  EXPOSE 22004
5
6  ENV JAVA_HOME=/root/.sdkman/candidates/java/current
7  ENV PATH="$JAVA_HOME/bin:$PATH"
8
9  RUN mkdir -p /config/sachsen
10 COPY data/sachsen/config.properties /config/sachsen/config.properties
```

```
11
12 COPY source/simulator/target/max-simulator-1.0.0.jar .
13 COPY source/simulator/start.sh .
14
15 ENTRYPOINT ["./start.sh"]
```

Listing 7.1: Dockerfile for building the simulator

We start by specifying a base image containing Java runtime to run our built solution as a `.jar` file. We declare what ports need to be exposed to the host network, but we will see later that this serves only documentation purpose. We perform some setup like specifying environment variables needed for running, and then copy needed the configuration. Lastly we copy the actual solution and the script for running it. The script simply calls the following command.

```
1 java -Xmx384m -Xdebug \
2      -Dspring.config.additional-location=file:config_cluster/ \
3      -Dcluster-map.ownId=10 \
4      -jar max-simulator-1.0.0.jar \
5      --simulator.configBase=config --simulator.configs=sachsen,claro
```

Listing 7.2: Java command for running the solution (the arguments are less important, but most importantly we specify configuration folders)

The reason why we extract the startup in to a script is so that in case someone wants manually run the solution they only need to run the script. We automate this process through a deploy stage job for each component that builds the Docker image and pushes it to the container registry of the solution repository.

```
1  deploy-container-simulator:
2    stage: deploy
3    image: gitlab:5005/tat/maxd/ubuntu-graal-docker:jdk21
4    services:
5      - name: docker:24.0.5-dind
6        alias: docker
7    variables:
8      DOCKER_TLS_CERTDIR: "/certs"
9    tags:
10     - docker-runner
11   before_script:
12     - docker info
13     - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
         $CI_REGISTRY
14   script:
15     - 'cd␣$SOURCE_MAVEN_PROJECT_DIR'
16     - cd ..
17     - docker build -t $SIMULATOR_IMAGE_TAG -f source/simulator/
         Dockerfile .
18     - docker push $SIMULATOR_IMAGE_TAG
```

```
19    needs: [ jar-maxd-core ]
```

Listing 7.3: Deploy simulator container job as part of the deploy stage

Since we are now using HTTPS, we also have to specify the location of the certificates needed for Docker. We begin by moving to the directory containing the source code, after which we build the actual image and push it. This job relies on a previous job that simply builds the whole project and persists only the needed target files needed for containerization.

7.1.4 Cloud testing

As we've defined previously, all the cloud tests will be located in a separate repository with its own pipeline. To trigger that pipeline from another project (solution repository) we must add another job that will just trigger the pipeline and whose result depends on the execution of the other pipeline.

```
1  cloud-test-pipeline:
2    stage: cloud-test
3    trigger:
4      project: cloud/cloud-tests
5      branch: "main"
6      strategy: depend
```

Listing 7.4: Dockerfile for building the simulator

The cloud test pipeline follows a similar structure as the prototype pipeline we developed. The difference is that in this case the cloud test will only perform tests, without any building or deployment. We reuse the testing workflow of the prototype pipeline by having a cloud-deploy, test, and cloud-cleanup jobs. Because of this, we will focus only the actual test defined in the Ansible playbook. For provisioning we used OpenTofu and spawn the following components:

- **VM 1**: Advanced Protection System
- **VM 2**: Plan Execution
- **VM 3**: Simulator
- **Network 1**: Network containing all VMs
- **Router**: Connecting the network to the external network (for Ansible to access)

In Ansible we make sure that all virtual machines contain the needed dependencies for running the solution. This includes Docker and other configuration files that depend on IP addresses of the virtual machines which we create using template file provider in OpenTofu.

```yaml
- name: Setup
  hosts: all
  become: yes
  gather_facts: no
  tasks:
    - name: Wait for the vms to boot
      pause:
        seconds: 45

    - name: SSH wait
      wait_for:
        port: 22
        timeout: 300
        state: started
      retries: 20
      delay: 5

    - name: Create the config cluster directory
      file:
        path: /config_cluster
        state: directory
        mode: '0755'

    - name: Copy the configuration
      copy:
        src: "{{ playbook_dir }}/config_cluster/application.yml"
        dest: "/config_cluster/application.yml"
        mode: '0755'

    ...

    - name: Install Docker packages
      package:
        name: docker-ce
        state: present

    - name: Ensure Docker service is enabled and running
      service:
        name: docker
        state: started
        enabled: yes
```

Listing 7.5: Ansible task for machine setup

We await a certain period of time to ensure that all the infrastructure is up and running, but despite this we make sure that SSH is running (and retrying in case the virtual machine is still not running). We create all the needed directories and then copy the configuration. `application.yml` contains information of all the components and their

information (IP address, ports they listen on). `opPlan-relative.json` is a file that contains operational plans (train travel path) needed for the simulator for testing. By specifying `hosts` to `all` we instruct Ansible to run the job on all the instances in the inventory. The ommited part marked with three dots contains some preparation for the Docker installation.

```
- name: Run sim
  hosts: sim_vm
  become: yes
  gather_facts: no
  vars:
    registry_url: "{{ lookup('env', 'CI_REGISTRY') }}"
    registry_username: "{{ lookup('env', 'CI_REGISTRY_USER') }}"
    registry_password: "{{ lookup('env', 'CI_REGISTRY_PASSWORD') }}"
  tasks:
    - name: Login to registry
      docker_login:
        registry_url: "{{ registry_url }}"
        username: "{{ registry_username }}"
        password: "{{ registry_password }}"

    - name: pull image
      docker_image:
        name: "gitlab:5005/tat/maxd/simulator-pipeline:latest"
        source: pull

    - name: run simulator
      docker_container:
        name: simulator
        image: "gitlab:5005/tat/maxd/simulator-pipeline:latest"
        env:
          REDIS_HOST: "{{ hostvars['aps_vm'].internal_ip }}"
        published_ports:
          - "8080:8080"
          - "22004:22004"
        network_mode: host
        detach:
          true
        volumes:
          - "/config_cluster:/config_cluster"
        state: started

    - name: wait for the container
      pause:
        seconds: 15
```

Listing 7.6: Ansible task for running the simulator

Since we will pull the Docker images to create the containers of each component, we need

to define various registry variables by accessing environment variables of the pipeline. After this we login to the registry, we pull the image for the simulator and then run it as a container. As previously mentioned exposing ports is only beneficial when the container isn't running in host network mode. However, we decided to still keep it as it serves documentation purposes, as each virtual machine needs to have a security group created containing all the ports that need to be exposed. We let the solution run for 15 seconds and then we print out the Docker logs, after which the container keeps running in the background as we detached it. Once started up we can perform the tests.

```
- name: Perform test
  hosts: sim_vm
  become: yes
  gather_facts: no
  tasks:
    - name: Start listening to aps events
      shell: "nohup␣curl␣-N␣http://{{␣hostvars['aps_vm'].internal_ip␣
          }}:8081/os/events-stream␣>␣/tmp/aps-events␣&"
      async: 0
      poll: 0

    - name: Start listening to pe events
      shell: "nohup␣curl␣-N␣http://{{␣hostvars['pe_vm'].internal_ip␣
          }}:8085/planExec/events>␣/tmp/pe-events␣&"
      async: 0
      poll: 0

    - name: Start the simulation in normal speed
      uri:
        url: "http://{{␣hostvars['sim_vm'].internal_ip␣}}:8080/
            simulator/run"
        method: POST
        headers:
          Content-Type: "application/json"
        body: '{"speed":␣"NORMAL"}'
        body_format: json

    - name: Load operational plan
      shell: "curl␣-X␣POST␣-d␣'@/claro/opPlan-relative.json'␣-H␣'
          Content-Type:␣application/json'␣'http://{{␣hostvars['sim_vm
          '].internal_ip␣}}:8080/tmsSimu/loadOpPlan'"
      async: 60
      poll: 0

    - name: Wait for the simulation to finish
      pause:
        minutes: 35
```

Listing 7.7: Ansible task for performing the test

As previously mentioned, we interact with the simulator to perform the tests. We start listening to PE events and APS events which will later serve for debugging and test outcome definition. We redirect all the events to files we will later expose as artifacts to the pipeline. We start by specifying the speed to `NORMAL` which starts the simulator (time begins passing). This is followed by sending the operational plan which will create a train and define a travel path for it. As currently only `NORMAL` speed is implemented in the distributed deployment we have to wait in real time for the train to travel to its destination for which we wait 35 minutes. The last step is to gather all the logs and declare the test as passing or failing.

```
- name: Get logs
  hosts: all
  become: yes
  gather_facts: no
  tasks:
    - name: Extract container name from hostname
      set_fact:
        log_name: "{{ inventory_hostname.split('_vm')[0] }}"

    - name: "Get docker logs"
      shell: "docker logs {{ log_name }} > /tmp/{{ log_name }}.logs"
      args:
        creates: "/tmp/{{ log_name }}.logs"

    - name: "Fetch log file to the local machine"
      fetch:
        src: "/tmp/{{ log_name }}.logs"
        dest: "{{ playbook_dir }}/{{ log_name }}.logs"
        flat: yes

    - name: "Fetch aps events"
      fetch:
        src: /tmp/aps-events
        dest: "{{ playbook_dir }}/aps-events"
        flat: yes
      when: inventory_hostname == 'sim_vm'

    - name: "Fetch pe events"
      fetch:
        src: /tmp/pe-events
        dest: "{{ playbook_dir }}/pe-events"
        flat: yes
      when: inventory_hostname == 'sim_vm'

- name: Check if test has failed or passed
  hosts: sim_vm
  become: yes
  gather_facts: no
```

```
39  tasks:
40    - name: Check if train is deleted
41      shell: cat /tmp/pe-events | grep "data:UtoCreated.*TRAIN_EOM\|
            data:TrainDeleted"
42      register: log_check
43      ignore_errors: true
44
45    - name: Found message
46      debug:
47        msg: "{{ log_check.stdout_lines }}"
48      when: log_check.rc != 0
49
50    - name: Fail the task if grep finds nothing
51      fail:
52        msg: "Pattern not found in the file."
53      when: log_check.rc != 0
54
55    - name: Found message
56      debug:
57        msg: "{{ log_check.stdout_lines }}"
```

Listing 7.8: Ansible tasks for fetching logs and test outcome declaration

Regardless of the test outcome we will gather logs which includes Docker logs of every component and event logs. We specify this task before declaring the test as failing or passing. Test outcome is defined by analyzing the PE events and the last event should be the train reaching the destination and being deleted, for which we utilize `grep` to match the pattern of the event. When fetching container logs we have to specify the name of the container which we can extract from the inventory hostname, which is also used to define conditional tasks that will be only run on the simulator virtual machine since it is the only one containing PE and APS events.

7.2 User survey

In order to better assess the applicability and usability of the pipeline we performed a survey amongst the developers in the innovation team. They were presented the pipeline implementation applied to integration tests performed in the innovation team, after which they were allowed to use it and ask any questions regarding it. We followed this by performing a survey in order to elicit feedback and impressions of the testing pipeline. The developers in the team were already experienced with performing these tests manually, but not automatically in the cloud. We purposely kept the survey short and concise consisting of only the most important questions relevant to our evaluation. The innovation team is a team of five, consisting of a manager and four developers, one of which is the author of this work and thus excluded from the survey. The rest of the developers took part in the survey.

7.2.1 Survey structure

The survey consisted of six questions divided into three sections. The first section serves as a way for the participants to consent to the usage of their information for the purposes of this work:

- **Q.1: Do you consent to the usage of the information given in this survey for the scientific purposes of the thesis? (Yes/No)**

The second section gathers information about how the system is being tested at the moment. Firstly, the participants explain the way it is tested, followed by a ranking of their satisfaction with the current way of testing, and lastly outlining problems they see with it.

- **Q.2: Please explain how the current testing of MBS is done (e.g. setup, simulator usage, testing, checking for correct behavior).**
- **Q.3: Please rate your experience of using the current test setup for MBS in terms of satisfaction (1-10, 1 being extremely satisfied, 10 being extremely unsatisfied).**
- **Q.4: What would you say are the problems of the current testing setup for MBS? Leave blank if you think there are no problems.**

The third, and the last section, consists of questions regarding the proposed pipeline for automatic testing, which they were introduced to before the survey took place.

- **Q.5: After seeing the cloud tests applied to MBS in action, do you think that the new automatic testing in the cloud improves upon the previous testing method? If yes, explain how.**
- **Q.6: Would you like to see more types of tests implemented in the cloud? If yes, name and explain them.**

7.3 Survey results

We will now present the results and findings from the survey. All the participants consented to the usage of their responses so all the participants are considered in the results. We will take a look at the answers of the second section containing questions Q.2, Q.3, Q.4 related to the current testing setup, followed by an analysis of the third section containing questions Q.5 and Q.6 related to the proposed automatic cloud tests.

7.3.1 Current testing setup

From the answers provided in this sections we gathered the following:

- **Q.2: Current testing setup**
  - Only unit tests and acceptance tests are automatic, while smoke testing is performed manually.
  - Each component (APS, PE, simulator) is tested in isolation using unit tests.
  - For integration tests, one developer states that they use scripts to send REST API requests, but the correctness is checked manually, via visual inspection.
  - Integration tests are performed through web UI where results are again checked via visual inspection.
- **Q.3: User satisfaction (Figure 7.2)**
  - The average score given amongst the participants was 4.67.
  - If we consider any rating of 5 and above to be satisfied, only one developer can be considered satisfied.
- **Q.4: Problems with current testing setup**
  - There are currently no tests in place that test the interactions of all the components, as well as the components being deployed on different machines.
  - Distributed deployment is not tested enough.
  - Currently available automatic tests do not provide sufficient coverage.

Figure 7.2: User satisfaction (Q.3) with regards to the current (manual) testing setup. Values below 5 are marked red, otherwise green.

7.3.2 Automatic cloud tests

Following conclusions can be derived from the answers in this section:

- **Q.5: Improvements**
  - All the participants believe that the automatic cloud tests improve upon the current testing setup.
  - Of the improvements, following were mentioned: automation of different scenarios and configurations, automatic integration tests, distributed deployment testing, testing with multiple actors.
- **Q.6: Future work**
  - Availability and scalability testing, component failure simulation
  - Two participants expressed interest in integration tests that focus on the interactions of the components

7.3.3 Survey conclusion

Taking the responses of the participants into account, we can clearly identify that manual tests result in mostly unsatisfied users. We can also see how there is much potential for automation especially in the aspect of integration tests. Cloud tests also offer a way of testing distributed deployment where each component is deployed to a different machine, something that is impossible to test locally.

7.4 Use case discussion

In order to conclude the evaluation we will now compare some attributes of both approaches for the Hitachi Rail use case. Before the pipeline was developed we performed the integration tests manually, the way it was previously, after which we implemented the pipeline for automating them, and lastly used the pipeline to achieve the same tests by simply making changes to the code and triggering them, thus practically using them. Taking both the implementation and the effort needed to use the pipeline we can compare it to the previous manual testing methods in regards to two aspect:

- **Testing duration:** Manually performing tests requires running all the required services (APS, PE, simulator) on the local machine, after which we access the web UI, we load the operational plan and begin observing. We act as the test oracle by expecting some outcome (train reaching destination) and the test ends there. We also then have to perform cleanup by stopping the execution of all the services. Currently, due to a bug in the distributed deployment, simulation speed can only be set to real-time and as such will take approximately the same amount of time for both the manual and automatic tests as both have to wait 35 minutes for the

train to reach the destination. However manual testing has a slight edge if we take into account the solution build time as it is faster on the developer machines. Moreover, should the simulation time bug be fixed we could set the speed to 8 times the normal speed which would be hard for a person to manually observe giving an edge to automatic tests that rely solely on events. **In the current state of the solution, manual tests take less time compared to automatic tests, taking into account the whole pipeline workflow.**

- **Testing effort:** To perform manual tests, first and foremost the user has to understand how to run the test, which components to run, how to perform the test, and lastly how to validate it. Even if we do not take this into account, but assume the test is performed by someone knowledgeable, the effort needed is far greater compared to the automatic tests that require no effort since they will be trigger and performed independently. **Compared to manual tests, automatic cloud test require far less effort, with only effort being the analysis of the pipeline result.**

- **Initial effort:** In contrast to the testing effort, we can also analyze the effort needed to setup the pipeline for automatic cloud tests in comparison to the manual tests. It is clear that test automation only happens after the initial tests are defined, and then migrated to the cloud. Of course, automatic cloud tests can be considered from the start, but still need to be manually defined in order to be automated. During the implementation phase of the Hitachi Rail use case, a lot of time was spent setting up the OpenStack project and the actual pipeline, before even defining a single test. **The initial effort needed for manual testing is far less compared to automatic cloud tests that incur a significant initial cost in terms of time.**

- **Hardware resources:** Manual tests are performed on the developer machine, they do not require additional hardware resources, still they occupy the developer resources during their execution. **For automatic cloud test, hardware resources are required in order to host the actual cloud, introducing additional costs in terms hardware.**

We can also perform some calculations[1]that can show the potential of automatic cloud tests, but we have to define some assumptions. The solution whose tests are being migrated to the cloud does not exhibit any bugs in the process of the migration, and the developer in charge of the migration already has knowledge of DevOps principles and tools (e.g. IaC), as well as access to an OpenStack project. In this case we can estimate that the migration process would last 10 working days for our use case. Since manual tests take about 40 minutes to perform (automatic tests require no manual intervention and thus last 0 minutes from the perspective of the developer in terms of effort), and for the three developers using them, we can assume they each perform one test a day, which for a working month accounts to 60 tests, resulting in 2400 minutes spent testing. If

we also assume that we invest 10 hours monthly on cloud test maintenance we subtract 600 minutes from the time spent testing resulting in 1800 minutes a month saved per month. Since we also estimated it takes about 10 working days to setup cloud tests, or 4800 minutes, dividing this by the minutes saved per month (2400 minutes) we get 2.67 as the amount of months needed to repay the time invested into test migration to the cloud. The time to repay would be larger if the manual tests took less time to perform. This means that once the simulation speed bug is resolved, time to repay would become larger. We can clearly see that automatic tests incur a large initial cost that can grow significantly depending on the developer experience, and the available infrastructure (OpenStack project, hardware resources for the cloud), but they act as a better long term option as they remove the testing effort.

Metric	Manual tests	Automatic tests
Testing duration[2]	–	+
Testing effort	–	+
Initial effort	+	–
Hardware resources	+	–

Table 7.1: Tabular pros/cons comparison between manual and automatic tests in terms of measured metrics. (+ means in favor of, - means worse)

Throughout the process of migrating the integration tests to the cloud a lot of previously unknown bugs were uncovered. Distributed deployment was barely tested before, and it has been uncovered that only normal simulation speed (real time speed) can be utilized in that case, which should result in faster tests once fixed and faster simulation can be performed (fastest being 8 times faster than normal). In the beginning the train did not reach the destination successfully because of other bugs uncovered which were later addressed and fixed. In some cases running the test manually on the developer machine would not work due to the lack of idempotency. The migration process showed that testing on the developer machine alone was not enough, and the integration tests performed before were inadequate for both solution logic and distributed deployment.

There are few things to consider when drawing conclusions from this use case, that may be specific to the use case. The test execution time will depend on the hardware resources of the cloud and of the developer machine running manual tests. Another important factor is that currently the simulation speed can only be done in real time (normal speed) due to bugs uncovered and once fixed will result in 8 times faster simulation time and thus test execution time. Also, in our case the developers where performing manual tests on their machines, and have not tried to physically imitate the environment where the

[1]The calculations performed were based on ROI (Return On Investement) calculations[60].
[2]Testing duration results assumes a case where the simulation speed bug is fixed.

actual solution would run. If that were the case there would be significantly more effort needed for manual testing as it would require time and additional costs in the setup of those machines, as well as maintenance.

However, analyzing the survey as well as the previously described comparison, we can conclude the following: **Automatic cloud tests introduce additional costs in terms of time and hardware resources during the setup and implementation phase, but significantly lower the effort of using the tests, while enabling previously impossible test scenarios.** In the end, depending on what the resources allow, manual tests might be significant in smaller cases, but automatic cloud tests act as a better long term option.

CHAPTER 8

Discussion

8.1 Discussion of research questions

We will now present each research question again and provide an answer to them.

- **RQ.1: What are the state-of-the-art open-source IaC and CI/CD technologies for enabling testing in the cloud using simulated environments?**

  In the Chapter 2 we've discussed various state-of-the-art technologies as well as technologies that serve as the predecessors and influencers of the state-of-the-art. We recognize Gitlab CI and Jenkins as the state-of-the-art CI/CD technologies that have garnered much attention in both the scientific world as well as in practice. For provisioning we outline OpenTofu and Pulumi, and for configuration management Ansible as well as Puppet and Chef as the state-of-the-art.

- **RQ.2: What are the requirements of a fully automated pipeline for testing in the cloud using simulated environments?**

  In the Chapter 4 we performed requirements elicitation in order to derive a set of requirements needed for implementing the fully automated pipeline, which covers requirements in terms of the pipeline, test structure, provisioning and configuration management. Depending on the use case the requirements provided may prove non-exhaustive but aim to provide better understanding of what is required of such a pipeline.

- **RQ.3: What is the architecture for a fully automated pipeline for testing in the cloud using simulated environments?**

  By researching the state-of-the-art as well as performing interviews with experts, we derived and used the following technologies: Gitlab CI for a CI/CD pipeline,

Ansible for configuration management, OpenTofu for provisioning and OpenStack for cloud computing.

- **RQ.4: What are the benefits and drawbacks of testing in the cloud using simulated environments compared to the traditional testing methods?**

  Throughout research, interviews, implementation and evaluation we've identified many benefits. Automation as the key concept of this work has showed much potential from automating the deployment, to configuration management, and finally to tests. By automating tests, any changes will be tested automatically regardless whether or not the developer would know how to perform those tests manually. By automating the whole deployment process, we remove the need for manual maintenance. Simplicity of IaC lowers the bar of entry for provisioning and configuring of infrastructure which allows the developers to describe all the needed hardware and software through code. As each pipeline runs in its own environment we can run many tests in parallel, only limited by actual hardware resources. As we deploy and cleanup all the infrastructure needed for every test, we ensure idempotency, which is hard to guarantee when running tests locally.

  Considering the drawbacks, one of the biggest is the complexity this approach introduces. By relying on multiple technologies utilizing cloud and IaC debugging and implementing spans multiple points. In addition, knowledge needed to properly apply these technologies and develop the pipeline slows down the transition process. Cloud testing also introduces additional costs in terms of hardware needed to host the cloud, as well as time needed to maintain it. Should one take the approach of a public cloud, costs of using the cloud have to be considered.

8.2 Limitations

In this section we will present and discuss some limiting factors that have influenced this work and its results.

Interview sample size

The interview (Chapter 5) performed included 5 participants who were eligible in terms of experience and knowledge to discuss and answer the interview questions. Although they all showed experience in previous companies, they represent only a group of people currently working under the same company. This meant that we were limited to only a certain number of participants that both had time for the interviews, and knowledge needed.

Resource limitations

One of the biggest limitations of exploring cloud computing is the hardware resources needed for hosting the cloud. In our work we were limited by the number of allowed

simultaneous virtual machines, networks, routers, as well as hardware resources like vCPUs and RAM. More concretely, 10 instances (where 4 were already used for setup and hosting of needed infrastructure), 5 networks, 5 routers and 2 floating IPs, as well as 80 vCPUs and 160GB of RAM. The biggest limiting factor was the number of instances and networks which prevented us from simulating a large number of users, or a very large network topology.

Concrete hardware simulation

Although we can specify vCPUs, RAM, and storage, we cannot specify underlying architecture of the processor. We cannot specify RAM speed, nor the type of storage device like HDD, or SSD. Because of this, we cannot guarantee fully that the virtual environment in the cloud completely replicates the client environment.

Survey sample size

As part of the evaluation chapter we conducted a survey within a small team, with three developers partaking in the survey. Although the results shown demonstrated general consensus regarding the state of manual tests and improvements of automatic tests, should the sample size be larger our arguments would have been even more prominent, and perhaps we would encounter outliers that argue against the migration of the tests to the cloud.

Use case choice

Hitachi Rail use case allowed us to evaluate the automatic cloud testing approach on a real life use case, however it only represents one use case. Even though we did not have to adjust the pipeline drastically, it would still be useful to explore how the pipeline would have to be adapted for other use cases that require more complex testing setup, or more complex types of tests. Also, our use case is within the railway signaling domain, however the business logic of the domain played less of a role during the implementation for this use case, but it would still be useful if we could explore other domains.

8.3 Lessons learned

In this section we will discuss lessons learned in the form of challenges we've encountered throughout the work on the thesis, which serves as a guide to potential difficulties one can expect by taking this approach. We believe by also outlining the difficulties of testing in the cloud we bring a more realistic picture of practical use of cloud testing.

Learning curve

An obstacle which can discourage some from automating and migrating their tests in the cloud is first and foremost the initial learning curve. Cloud computing provides

many challenges, and hosting your own cloud becomes an immense challenge for any beginner. Debugging requires knowledge of underlying hardware, network stack, as well as knowledge of host operating system. Throughout the pipeline implementation, much time was spent debugging why certain components where not interacting correctly, which in most cases required thorough reading of various documentation, of which many assume a prerequisite knowledge of DevOps principles and cloud computing.

Setup

Due to the complexity of the pipeline architecture, as well as many components coexisting, much time was spent setting up the actual infrastructure. As we opted to host everything on our own, we had to setup Gitlab, Gitlab Runners, followed by creating our own Docker image for pipeline execution containing all the needed dependencies (Docker, Ansible, Terraform, etc.). This was followed by configuring and authorizing each component to be able to access to the cloud resources as well as Gitlab resources. Only then could we proceed to implement the pipeline logic.

CHAPTER 9

Conclusion

Firstly we introduced the topic and the goals of this work, which we've followed by a chapter dedicated to related work. In it, we provided an overview of the most important concepts and practices that have paved the way for cloud testing. We've outlined how these principles spawned various technologies that have enabled us to implement the pipeline. We concluded the chapter by providing a literature review providing insight in to what research has been performed in DevOps, CI/CD, IaC, as well as cloud computing. Moreover we performed various interviews with experts in Hitachi Rail, eliciting knowledge and summarizing it, and in conclusion deriving a technology stack which we used for the implementation of the pipeline. In order to better understand what the pipeline needs to achieve, we derived a requirement specification. In the implementation chapter we covered step by step how we achieved the desired pipeline which we applied to a use case in the railway industry. We evaluated the applicability of cloud tests using this use case, which helped us outline the benefits and challenges. We compared cloud testing to the traditional testing methods and performed a survey to gather user experience of using cloud tests.

9.1 Future work

We will now provide various research avenues for future work in the field of testing in the cloud. In this paper we focused on open-source technologies, but this leaves a lot of state-of-the-art technologies unexplored. Amazon Web Services, Google Cloud Platform, as well as Microsoft Azure are cloud computing technologies worth exploring, that offer their ways of provisioning and configuration of cloud infrastructure (e.g. AWS CloudFormation). This work only considers a subset of open-source technologies for the pipeline implementation, and as such it is worth exploring how this can be achieved with other technologies and then compared. It is worth exploring what open-source technologies do various companies utilize by interviewing more experts from different

companies. The concept of testing in the cloud can be applied to various use cases, which can serve as a starting point for more case study research which would serve as an argument for migrating to the cloud and the applicability of testing in the cloud.

9.2 Concluding remarks

Testing in the cloud has clearly demonstrated benefits, however a lot of time and effort is needed in the initial setup phase. Migrating tests to the cloud requires knowledge of many technologies and layers, which severely raises the initial cost and in some cases it might outweigh the benefits. Still, as advances are made in cloud computing and infrastructure as code, we might see a flattening in the learning curve, as well as a reduction in setup time. In end effect, even simple tests running in the cloud will free up developer resources by moving the testing process away from the developer machine, as well as free up time by removing human intervention.

Unless exact hardware is needed, and enough time and resources can be invested, testing in the cloud shows much potential and provides many benefits to its user. Most notably automation, as the cornerstone of infrastructure as code, alleviates the burden of testing as an inevitable part of the development cycle.

Overview of Generative AI Tools Used

List of Figures

List of Tables

Bibliography

[1] About Cookbooks — docs.chef.io. https://docs.chef.io/cookbooks/. [Accessed 03-11-2024].

[2] About Recipes — docs.chef.io. https://docs.chef.io/recipes/. [Accessed 03-11-2024].

[3] All modules &x2014; Ansible Documentation — docs.ansible.com. https://docs.ansible.com/ansible/2.8/modules/list_of_all_modules.html. [Accessed 02-11-2024].

[4] Chef Infra Overview — docs.chef.io. https://docs.chef.io/chef_overview/. [Accessed 03-11-2024].

[5] Chef Software DevOps Automation Solutions | Chef — chef.io. https://www.chef.io/. [Accessed 03-11-2024].

[6] Command: apply | Terraform | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/terraform/cli/commands/apply. [Accessed 02-11-2024].

[7] Command: destroy | Terraform | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/terraform/cli/commands/destroy. [Accessed 02-11-2024].

[8] Command: plan | Terraform | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/terraform/cli/commands/plan. [Accessed 02-11-2024].

[9] Concepts — pulumi.com. https://www.pulumi.com/docs/iac/concepts/. [Accessed 03-11-2024].

[10] Download and install GitLab — about.gitlab.com. https://about.gitlab.com/install/. [Accessed 22-10-2024].

[11] Downstream pipelines | GitLab — docs.gitlab.com. https://docs.gitlab.com/ee/ci/pipelines/downstream_pipelines.html?tab=Multi-project+pipeline. [Accessed 08-11-2024].

[12] Get started with GitLab CI/CD | GitLab — docs.gitlab.com. https://docs.gitlab.com/ee/ci/. [Accessed 02-11-2024].

[13] GitHub - hashicorp/terraform: Terraform enables you to safely and predictably create, change, and improve infrastructure. It is a source-available tool that codifies APIs into declarative configuration files that can be shared amongst team members, treated as code, edited, reviewed, and versioned. — github.com. https://github.com/hashicorp/terraform. [Accessed 03-10-2024].

[14] GitLab Runner | GitLab — docs.gitlab.com. https://docs.gitlab.com/runner/. [Accessed 02-11-2024].

[15] Homepage | Ansible Collaborative — ansible.com. https://www.ansible.com/. [Accessed 02-11-2024].

[16] How to build your inventory &x2014; Ansible Community Documentation — docs.ansible.com. https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html. [Accessed 02-11-2024].

[17] Identity API v3 (CURRENT) &x2014; keystone documentation — docs.openstack.org. https://docs.openstack.org/api-ref/identity/v3/index.html. [Accessed 03-11-2024].

[18] Install GitLab Runner | GitLab — docs.gitlab.com. https://docs.gitlab.com/runner/install/. [Accessed 22-10-2024].

[19] Introduction: A Bit of OpenStack History &x2014; OpenStack Project Team Guide documentation — docs.openstack.org. https://docs.openstack.org/project-team-guide/introduction.html. [Accessed 03-11-2024].

[20] Jenkins — jenkins.io. https://www.jenkins.io/. [Accessed 02-11-2024].

[21] Manifesto | OpenTofu — opentofu.org. https://opentofu.org/manifesto/. [Accessed 03-10-2024].

[22] MPL 2.0 FAQ — mozilla.org. https://www.mozilla.org/en-US/MPL/2.0/FAQ/. [Accessed 03-10-2024].

[23] Networking Service APIs &x2014; Networking API Reference documentation — docs.openstack.org. https://docs.openstack.org/api-ref/network/. [Accessed 03-11-2024].

[24] Open Source Cloud Computing Infrastructure - OpenStack — openstack.org. https://www.openstack.org/. [Accessed 03-11-2024].

[25] OpenStackClient &x2014; OpenStack Command Line Client 7.2.2.dev1 documentation — docs.openstack.org. https://docs.openstack.org/python-openstackclient/latest/. [Accessed 03-11-2024].

[26] os_server &x2013; Create/Delete Compute Instances from OpenStack &x2014; Ansible Documentation — docs.ansible.com. https://docs.ansible.com/ansible/2.8/modules/os_server_module.html. [Accessed 19-10-2024].

[27] Packaging Python Projects - Python Packaging User Guide — packaging.python.org. https://packaging.python.org/en/latest/tutorials/packaging-projects/. [Accessed 24-10-2024].

[28] Pipeline Syntax — jenkins.io. https://www.jenkins.io/doc/book/pipeline/syntax/#compare. [Accessed 02-11-2024].

[29] Providers - Configuration Language | Terraform | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/terraform/language/providers. [Accessed 20-10-2024].

[30] Pulumi - Infrastructure as Code, Secrets Management, and AI — pulumi.com. https://www.pulumi.com/. [Accessed 03-11-2024].

[31] pulumi preview — pulumi.com. https://www.pulumi.com/docs/iac/cli/commands/pulumi_preview/. [Accessed 03-11-2024].

[32] pulumi up — pulumi.com. https://www.pulumi.com/docs/iac/cli/commands/pulumi_up/. [Accessed 03-11-2024].

[33] Puppet Infrastructure & IT Automation at Scale | Puppet by Perforce — puppet.com. https://www.puppet.com/. [Accessed 03-11-2024].

[34] Puppet language overview — puppet.com. https://www.puppet.com/docs/puppet/7/intro_puppet_language_and_code. [Accessed 03-11-2024].

[35] Run your CI/CD jobs in Docker containers | GitLab — docs.gitlab.com. https://docs.gitlab.com/ee/ci/docker/using_docker_images.html#determine-your-docker_auth_config-data. [Accessed 08-11-2024].

[36] Stacks — pulumi.com. https://www.pulumi.com/docs/iac/concepts/stacks/. [Accessed 03-11-2024].

[37] State | Terraform | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/terraform/language/state. [Accessed 02-11-2024].

[38] Syntax - Configuration Language | Terraform | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/terraform/language/syntax/configuration. [Accessed 02-11-2024].

[39] Terraform by HashiCorp — terraform.io. https://www.terraform.io/. [Accessed 20-10-2024].

[40] Terraform Registry — registry.terraform.io. https://registry.terraform.io/providers/terraform-provider-openstack/openstack/latest/docs. [Accessed 02-11-2024].

[41] Using external secrets in CI | GitLab — docs.gitlab.com. https://docs.gitlab.com/ee/ci/secrets/. [Accessed 08-11-2024].

[42] Using Jenkins agents — jenkins.io. https://www.jenkins.io/doc/book/using/using-agents/. [Accessed 02-11-2024].

[43] What Are IaaS, PaaS and SaaS? | IBM — ibm.com. https://www.ibm.com/topics/iaas-paas-saas. [Accessed 02-11-2024].

[44] What is Continuous Delivery? – Amazon Web Services — aws.amazon.com. https://aws.amazon.com/devops/continuous-delivery/. [Accessed 01-11-2024].

[45] et al Ansible Collaborative. How Ansible works — ansible.com. https://www.ansible.com/how-ansible-works/. [Accessed 02-11-2024].

[46] Srikanth Baride and Kamlesh Dutta. A cloud based software testing paradigm for mobile applications. *ACM SIGSOFT Software Engineering Notes*, 36(3):1–4, 2011.

[47] Antonia Bertolino, Guglielmo De Angelis, Micael Gallego, Boni García, Francisco Gortázar, Francesca Lonetti, and Eda Marchetti. A systematic review on cloud testing. *ACM Computing Surveys (CSUR)*, 52(5):1–42, 2019.

[48] Shubham Bidkar, SL Patil, and Pravin Shinde. Virtual ecu development for vehicle diagnostics software testing using uds protocol. In *2021 Asian Conference on Innovation in Technology (ASIANCON)*, pages 1–6. IEEE, 2021.

[49] Todd Bishop. Silicon Valley mainstay NEA leads $37.5M investment in Seattle cloud startup Pulumi — geekwire.com. https://www.geekwire.com/2020/silicon-valley-mainstay-nea-leads-37-5m-investment-seattle-cloud-startu [Accessed 03-11-2024].

[50] Mark Burgess. markburgess.org. https://markburgess.org/papers/cfengine_history.pdf, 1993. [Accessed 19-10-2024].

[51] Artur Cepuc, Robert Botez, Ovidiu Craciun, Iustin-Alexandru Ivanciu, and Virgil Dobrota. 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)*, pages 1–6, 2020.

[52] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015.

[53] Armon Dadgar. HashiCorp adopts Business Source License — hashicorp.com. https://www.hashicorp.com/blog/hashicorp-adopts-business-source-license. [Accessed 03-10-2024].

[54] A Dhanapal and P Nithyanandam. An openstack based cloud testbed framework for evaluating http flooding attacks. *Wireless Networks*, 27(8):5491–5501, 2021.

[55] M.P. Dhanveer Prakash and Nidhi Sharma. The convergence of devops and cloud computing: A redefining software development. In *2023 Seventh International Conference on Image Information Processing (ICIIP)*, pages 800–805, 2023.

[56] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.

[57] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.

[58] Nick Fox. Using interviews in a research project. *The NIHR RDS for the East Midlands/Yorkshire & the Humber*, 26, 2009.

[59] B. Cameron Gain. Could Terraform Return to Open Source Under IBM's Ownership? — thenewstack.io. https://thenewstack.io/with-ibms-open-source-roots-terraform-could-be-free-again/. [Accessed 19-10-2024].

[60] Omar Galeano. What is the ROI of my test automation? — medium.com. https://medium.com/slalom-build/what-is-the-roi-of-my-test-automation-10ae7bf0d9ed. [Accessed 04-12-2024].

[61] Manu Gupta, Mandepudi Nobel Chowdary, Sankeerth Bussa, and Chennupati Kumar Chowdary. Deploying hadoop architecture using ansible and terraform. In *2021 5th International Conference on Information Systems and Computer Networks (ISCON)*, pages 1–6. IEEE, 2021.

[62] HashiCorp. HashiCorp | The Infrastructure Cloud Company — hashicorp.com. https://www.hashicorp.com/license-faq. [Accessed 03-10-2024].

[63] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 426–437, New York, NY, USA, 2016. Association for Computing Machinery.

[64] https://about.gitlab.com/about. GitLab 8.0 released with new looks and integrated CI! — about.gitlab.com. https://about.gitlab.com/releases/2015/09/22/gitlab-8-0-released/. [Accessed 02-11-2024].

[65] Samuel Kadima. Ansible: History Of Ansible — kadimasam. https://medium.com/@kadimasam/ansible-history-of-ansible-4b16208c9188. [Accessed 02-11-2024].

[66] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Comput. Surv.*, 52(6), November 2019.

[67] Hareton KN Leung and Lee White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301. IEEE, 1990.

[68] A Malini, N Venkatesh, K Sundarakantham, and S Mercyshalinie. Mobile application testing on smart devices using mtaas framework in cloud. In *International Conference on Computing and Communication Technologies*, pages 1–5. IEEE, 2014.

[69] Abbas Mehdi and Ranjan Walia. Terraform: Streamlining infrastructure deployment and management through infrastructure as code. In *2023 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, pages 851–856, 2023.

[70] Varsha Mittal, Lata Nautiyal, and Mohit Mittal. Cloud testing-the future of contemporary software testing. In *2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS)*, pages 131–136. IEEE, 2017.

[71] Bashir Mohammed and Mariam Kiran. Analysis of cloud test beds using opensource solutions. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 195–203. IEEE, 2015.

[72] Jaison Paul Mullerikkal and Yedhu Sastri. A comparative study of openstack and cloudstack. In *2015 Fifth International Conference on Advances in Computing and Communications (ICACC)*, pages 81–84, 2015.

[73] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[74] Sriniketan Mysari and Vaibhav Bejgam. Continuous integration and continuous deployment pipeline automation using jenkins ansible. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, pages 1–4, 2020.

[75] Aaron Paradowski, Lu Liu, and Bo Yuan. Benchmarking the performance of openstack and cloudstack. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 405–412, 2014.

[76] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108:65–77, 2019.

[77] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.

[78] Mojtaba Shahin, Muhammad Ali Babar, Mansooreh Zahedi, and Liming Zhu. Beyond continuous delivery: An empirical investigation of continuous deployment challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 111–120, 2017.

[79] Andra-Flavia Sicoe, Robert Botez, Iustin-Alexandru Ivanciu, and Virgil Dobrota. Fully automated testbed of cisco virtual routers in cloud based environments. In *2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, pages 49–53, 2022.

[80] Charanjot Singh, Nikita Seth Gaba, Manjot Kaur, and Bhavleen Kaur. Comparison of different ci/cd tools integrated with cloud platform. In *2019 9th International Conference on Cloud Computing, Data Science Engineering (Confluence)*, pages 7–12, 2019.

[81] Neelam Singh, Aman Singh, and Vandana Rawat. Deploying jenkins, ansible and kubernetes to automate continuous integration and continuous deployment pipeline. In *2022 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, pages 1–5, 2022.

[82] Mitesh Soni. End to end automation on cloud with build pipeline: the case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery. In *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 85–89. IEEE, 2015.

[83] Vamsi Krishna Thatikonda. Beyond the buzz: A journey through ci/cd principles and best practices. *European Journal of Theoretical and Applied Sciences*, 1(5):334–340, 2023.

[84] Adriano Vogel, Dalvan Griebler, Carlos A. F. Maron, Claudio Schepke, and Luiz Gustavo Fernandes. Private iaas clouds: A comparative analysis of opennebula, cloudstack and openstack. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 672–679, 2016.

[85] William Voorsluys, James Broberg, and Rajkumar Buyya. Introduction to cloud computing. *Cloud computing: Principles and paradigms*, pages 1–41, 2011.

[86] Shruti Yadav, Rajaram T Ugale, and Ritesh Goyal. Development of virtual test environment for vehicle level simulation. In *2023 3rd Asian Conference on Innovation in Technology (ASIANCON)*, pages 1–6. IEEE, 2023.

APPENDIX A

Setup details

A.1 Gitlab Secrets

In order to authorize the runner to perform certain actions, we need to provide it with certain authorization and authentication credentials. One approach would be to use a separate external secret manager [41]. For the purposes of this work we took another approach by utilizing masked variables. These can be defined in Gitlab CI/CD Variables section. Masked variables work only on non-whitespace text, so we encode the text using base64 and decode it in the pipeline. The following variables where utilized:

- **SSH keys**: We can store both the private and the public key for connecting to the cloud instances using Ansible. We defined two variables: **`SSH_PRIVATE_64`**, **`SSH_PUBLIC_64`**
- **Clouds yaml**: Terraform/OpenTofu use clouds.yaml to authenticate to OpenStack in order to deploy and provision infrastructure. We store this file as a base64 encoded text in the variable **`CLOUDS_YAML_64`**
- **Docker authentication**: To ensure that docker can login to the Gitlab registry to access container images we utilize **`DOCKER_AUTH_CONFIG`** [35] to store the contents of the Docker Json config.

A.2 Terraform/OpenTofu providers

Each time we call `tofu init/terraform init` we download all the needed providers defined in `providers.tf`:

```
terraform {
  required_providers {
    openstack = {
      source = "terraform-provider-openstack/openstack"
      version = "2.1.0"
    }
     template = {
      source = "hashicorp/template"
      version = "2.2.0"
    }
    local = {
      source = "hashicorp/local"
      version = "2.5.2"
    }
  }
}
```

Listing A.1: Specifying providers needed for Terraform/OpenTofu project

In a pipeline, it makes no sense to repeatedly download all the needed providers every single time when the pipeline is run. For this purpose, we can manually download each provider, by copying the resulting content of `tofu init`/`terraform init` into the repository. The next step is then to make Terraform/OpenTofu aware of this. In our case using OpenTofu we define a file `.tofurc` with the following content:

```
provider_installation {
  filesystem_mirror {
    include = ["registry.opentofu.org/*/*"]
    path    = "./providers"
  }
}
```

Listing A.2: Specifying providers needed for Terraform/OpenTofu project

Finally we make the OpenTofu aware of this by setting the environment variable **`TF_CLI_CONFIG_FILE`** to the path to `.tofurc`.

A.3 Deployment

Delivering the actual solution to the client can be done in various ways. This of course is done in agreement between the development team and the client. If the developers have access to the machine(s) that will run the solution on the client side one can connect to it using Ansible and install the needed software. Another way would be to build the solution as a Docker container and store it in a registry accessible to the client. Gitlab offers both a container registry and a release through artifacts. In our prototype pipeline we decided on exposing the built solution as an artifact available under Releases tab of the repository:

```yaml
deploy:
  stage: deploy
  before_script:
    - cd poc/python
  script:
    - python3 -m build
  artifacts:
    paths:
      - poc/python/dist
    expire_in: 2 days
  release:
    name: 'Addition Endpoint Version $CI_COMMIT_TAG'
    tag_name: '$CI_COMMIT_TAG'
    description: "Addition endpoint release"
    assets:
      links:
        - name: 'addition_endpoint'
          url: '$CI_SERVER_URL/$CI_PROJECT_PATH/-/jobs/$CI_JOB_ID/
              artifacts/download'
  only:
    - tags
```

Listing A.3: Deploy stage of the prototype pipeline

We build the solution, persist it by marking it as an artifact, and define the release by naming it, associating it with a tag (releases can only be done through tags) and defining the assets. The result will be a release that allows downloading of this artifact. To save space we limit the artifact to 2 days, but this can of course be changed to not expire at all.

A.4 Pipeline container image

If we run a Gitlab CI pipeline using Docker executor we need to provide an image that will be used for creating the container which will act as the pipeline environment. If we chose `Ubuntu:latest` we would have to install all the necessary dependencies (Ansible, Terraform, etc.) within the pipeline which would slow it down. For that we can create a separate repository that will have its own pipeline just for building the image that will be used in our cloud testing pipeline. We define a pipeline as follows:

```yaml
stages:
  - build

variables:
  IMAGE_NAME: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG

build-docker-image:
  stage: build
```

```
  image: docker:latest
  services:
    - docker:dind
  script:
    - echo $CI_REGISTRY
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER
        --password-stdin $CI_REGISTRY
    - docker build -t $IMAGE_NAME .
    - docker push $IMAGE_NAME

  only:
    - main
    - tags
```

Listing A.4: Gitlab CI pipeline for building a Docker image

Now all we have to do is define a `Dockerfile` as a recipe for creating the image. The condensed version is presented as follows:

```
FROM ubuntu:latest

ENV http_proxy="http://proxy:8889"
ENV https_proxy="http://proxy:8889"
ENV no_proxy="localhost,127.0.0.1,.gts"

RUN apt-get update && \
    apt-get install -y software-properties-common \
        apt-transport-https \
        ca-certificates \
        curl \
        ... && \

    apt-get clean

CMD ["/bin/bash"]
```

Listing A.5: DockerFile for building the Docker image used for running the pipeline