BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

INFRASTRUCTURE AS CODE IN AGILE SOFTWARE DEVELOPMENT

INFRASTRUKTURA JAKO KÓD V AGILNÍM VÝVOJI SOFTWARE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **VOJTĚCH HROMÁDKA**
AUTOR PRÁCE

SUPERVISOR **RNDr. MAREK RYCHLÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2019/2020

Bachelor's Thesis Specification

Student: **Hromádka Vojtěch**
Programme: Information Technology
Title: **Infrastructure as Code in Agile Software Development**
Category: Information Systems

Assignment:

1. Study Infrastructure as Code (IaC) technologies (e.g., Terraform, Chef, Ansible, Cloud Formation, Google Deployment Manager), evaluate and compare these technologies. Make yourself familiar with Continuous Delivery and Continuous Integration (CI/CD) concepts, technologies, and applications in agile software development.
2. Choose one IaC technology and describe its utilisation and possible issues in CI/CD in agile development using a Git code repository.
3. Design an agent which controls concurrent access to infrastructure resources in a cloud and prevents collisions of concurrent IaC deployments.
4. After consulting with the supervisor, implement the agent and demonstrate its usage in appropriate examples.
5. Describe, evaluate and publish the results as an open source.

Recommended literature:

- Yevgeniy Brikman. Terraform: Up & Running: Writing Infrastructure as Code. 2nd ed. O'Reilly Media, 2019. ISBN 1492046876.
- Gene Kim, Jez Humble, Patrick Debois, John Willis. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution, 2016. ISBN 194278807X.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Rychlý Marek, RNDr., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: July 31, 2020
Approval date: October 21, 2019

Abstract

This thesis is focused on the usage of infrastructure as code in agile software development. Concepts such as continuous integration and delivery, DevOps are analyzed. Further cloud environments are analyzed. In this work are compared different infrastructure as code tools. For the prevention of possible issues in using infrastructure as code software was designed. The software purpose is to control concurrent access to infrastructure creation with a tool called Terraform. The software was then is for experiments. The first experiment demonstrates that workflow with Terraform agent is behaving correctly. The second experiments demonstrate control of concurrent access to infrastructure creation.

Abstrakt

Tato bakalářská práce je zaměřena na využívání infrastruktury jako kódu v agilním vývoji software. Rozebírá další obvyklé koncepty, které jsou používány při agilním vývoji mezi které patří DevOps, kontinuální integrace a doručování. Dále je zaměřena na využití cloudu a na porovnávání jednotlivých nástrojů využívaných v infrastruktuře jako kódu. Pro prevenci možných problémů při využívání infrastruktury jako kódu byl navržen software, který má za účel kontrolovat souběžný přístup k vytváření infrastruktury s nástrojem Terraform. S tímto softwarem byly následně provedeny dva experimenty. První experiment demonstruje zdali lze uplatnit navrhovaný pracovní postup se softwarem, druhý experiment demonstruje správnost řešení při souběžném přístupu.

Keywords

Infrastructure as code, Agile development, Terraform, DevOps, Cloud

Klíčová slova

Infrastruktura jako kód, Agilní vývoj, Terraform, DevOps, Cloud

Reference

HROMÁDKA, Vojtěch. *Infrastructure as Code in Agile Software Development.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Rozšířený abstrakt

Využívání agilních metodik ve vývoji software roste čím dál tím více na popularitě. Vývojářské společnosti jsou schopni díky těmto metodám rychleji reagovat na požadavky od zadavatele, a tím přizpůsobit software dle nejnovějších potřeb. Obvyklou součástí agilního vývoje spočívá v přijetí DevOps kultury, která významně pomáhá urychlit proces doručování v podobě vytváření automatizovaných procesů jako je kontinuální integrace a kontinuální doručování.

Nedílnou součástí DevOps kultury a agilního vývoje je využívání nástrojů za účelem vytváření infrastruktury jako kódu. Infrastruktura jako kód umožňuje abstrakci samotného hardware do formy kódu jako je tomu zvykem při vytváření software. Pro vyváření infrastruktury jako kódu existuje několik nástrojů, které podporují vytváření infrastruktury na virtuálních strojích nebo nástroje které zajišťují prostředky od cloudových poskytovatelů. Tyto nástroje se obvykle dají zakomponovat do autimazitovaných rutin jako je kontinuální doručování.

Tato práce se zaměřuje na uvedení čtenáře do problematiky DevOps a praktických využití metod, které zrychlují vývoj a doručování softwaru zejména z pohledu vytváření infrastruktury s pomocí infrastruktury jako kódu.

Cílem této práce je prozkoumat možnosti infrastruktury jako kódu a popsat možný výskyt problémů při využívání daných nástrojů, poté navrhnout agenta, který zabrání kolizím při souběžném vytváření infrastruktury v cloudovém prostředí.

Hlavní částí celé práce je návrh a implementace serverového agenta, který je integrovaný do cloudové služby, tak aby byl schopný kontrolovat souběžný přístup k změnám infrastruktury. Pro vytvoření takového agenta je nutné zvolit nad kterým nástrojem bude pracovat. Prozkoumat jak daný nástroj pracuje, jaký je jeho obvyklý pracovní postup a ten potom zapouzdřit a vylepšit o požadované funkce.

Pro účely této práce je vybrát Terraform, který se jeví jako univerzální nástroj infrastruktury jako kódu. Následně jsou navrhnuty vylepšení pracovního postupu s daným nástrojem a to tak, že by měli zlepšit týmovou spolupráci. Navržený agent spolupracuje s verzovacím systémem GitHub tak, že pokaždé při vytváření nové verze infrastruktury, agent zajistí aby byla vytvořena nejnovější verze na základě posledních změn na GitHubu.

To všechno je na závěr práce naimplementováno a agent je nasazený do cloudu. Určité aspekty související s agentem jsou integrovány do cloudového prostředí pro správnou funkcionalitu celého programu. Druhá implementovaná část je klientská část programu, která je schopná komunikovat s agentem pomocí volání API.

Jako poslední bod práce jsou provedeny náležité experimenty, které demonstrují funkčnost softwaru a zároveň vysvětlují využití v praxi. První experiment má na starost zjištění základních požadavků na software jako je navržené zlepšení týmové spolupráce. Druhý experiment zobrazuje funkčnost programu při souběžném pokusu o vytvoření infrastrukturu.

Infrastructure as Code in Agile Software Development

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of RNDr. Marek Rychlý, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Vojtěch Hromádka
July 30, 2020

Acknowledgements

First I would like to thank my supervisor, RNDr. Marek Rychlý, Ph.D. For his willingness and his advice while creating this work. Also, I would like to thank Ing. Peter Malina from FlowUp that he helped me to put together the assignment of this thesis.

Contents

1 Introduction **2**

2 Agile development **3**
2.1 DevOps 4
2.2 Continuous integration and delivery 5
2.3 Team collaboration 7
2.4 Containerization 8
2.5 Cloud vs on-premise 9
2.6 Cloud-native 9
2.7 Cloud Providers 11

3 Infrastructure as a code **13**
3.1 Existing Infrastructure as Code Tools 14
3.2 Cloud specific IaC tools 15
3.3 Research on similar existing solutions 16

4 Design **18**
4.1 Terraform workflow 18
4.2 Designing the agent 19

5 Implementation **23**
5.1 Server Side 23
5.2 Client 26
5.3 Securing connection 26
5.4 Cloud configuration and deployment 27

6 Experiments **28**
6.1 Experimment 1 28
6.2 Experimment 2 30

7 Conclusion **32**

Bibliography **33**

A Content of the storage medium **35**

Chapter 1

Introduction

Software development methods are growing increasingly with a passion for agile development. The agile concept is focused on fast reaction to changes during development and fast delivery of new versions of the software even with small incremental changes.

Chapter 2 is about brief insight into agile and DevOps culture, about its utilization and common practices that are adopted by these cultures and how cloud is taking place in the current market and its benefits compared to on-premise solutions.

For agile development, proper tooling must be chosen to be able to deliver fast, with confidence and without errors. A Teams first steps to practising agile are usually utilising continuous integration and continuous delivery tools to build, test and deploy applications, but it does not end there.

In the actual world where cloud computing is increasingly getting popular, enterprises adapting tooling in form of infrastructure as code which allows abstract the physical layer of infrastructure. Infrastructure as code allows creating multiple environments of a project without the challenge to manage them all manually. A closer look into infrastructure as code practices is in Chapter 3.

This thesis focuses on practical aspects of using infrastructure as code in agile development and cloud environment. In Chapter 4.2 is designed agent that should improve team collaboration and allow control of concurrent access of creation infrastructure with Terraform tool to prevent collisions.

The motivation behind creating software that can control concurrent access to the infrastructure changes is to help enterprise teams to collaborate better without the need of explicit communication of new changes and also removing the urge of having dedicated team members to manage this kind of operations.

Chapter 5 focuses on the implementation of the Terraform concurrent agent and describes the approach that is used to create the application and how it is integrated into Google Cloud.

Last Chapter 6 describes testing and experimenting with the final application in the real scenarios and shows how the Terraform concurrent agent could be utilized in the development of larger projects.

Chapter 2

Agile development

Agile development is one of many development methodologies. This type of methodology is built on principles like simple design, continuous delivery, self-organizing teams and face-to-face communication, fast response. These principles are derived from four core agile values.

Composition of four agile values:

- **Individuals and interactions** over processes and tools
- **Working software** over extensive documentation
- **Collaboration with customer** over contract negotiation
- **Responding to change** over following a plan

In agile development value on the left side is more important than values on the right. However, it does not mean that values on the right side are not important. According to the set of these values twelve agile principles were proposed. These principles enhance the importance of agility in software development.

Some principles derived from these values are improved by using a cloud environment for development. Such as scalability, providing infrastructure (both hardware and software), fast delivery mechanisms, lowering cost and increasing software quality. In the bigger picture cloud computing affects agile software development with increasing prominence [22].

Agile software development has various methods and since general talk about it may not give a clear idea of how agile development works, Scrum is stated as the most popular agile framework.

Scrum

Scrum is defined as a flexible, holistic product, a development strategy where developers work as a unit to reach a common goal. One development cycle is called Sprint. Sprints are usually no long-term plans that have an elected amount of features that are implemented in one development cycle. After every sprint, Sprint Planning is arranged to prioritize the features. Sprints are created from Sprint Backlogs which works as a to-do list.

In Scrum daily meetings are held. Each team member should be prepared and share answers to three basic questions.

- What did the member yesterday do that contributed to sprint goal?

- What does the member plans to do today?
- Are there any difficulties that can prevent the member from contributing?

After each iteration, team members are part of a Retrospective meeting where they share and identify lessons and improvements for the next sprints [22].

With this simple example of how scrum works, it is safe to say that in the modern world agile development is a great way to work on projects for customers that are driven by fast-changing demand on the market as agile offers solutions for certain problems.

2.1 DevOps

In the first place, DevOps is a culture, not a specific method on how to approach an issue. However, there exist tactics that should help to create own methods which could shorten operations of software design changes [1].

DevOps is a culture and a mindset of people practising it. For most cases, that culture is about trust, team empowerment and cooperation. It also means DevOps is open to learning new things and finding solutions [16].

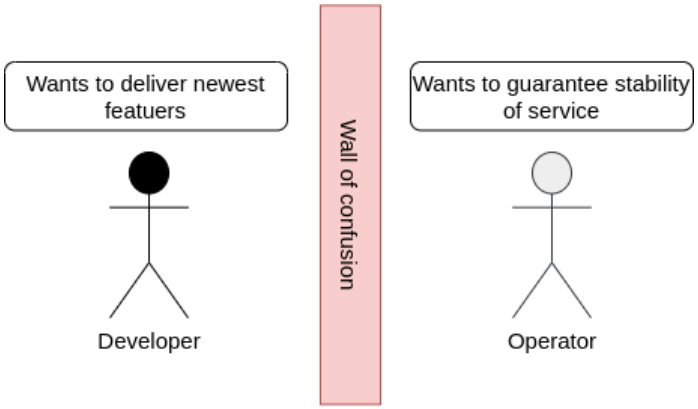

Figure 2.1: Misunderstanding is frequent event in divided teams

Dividing software development team into the development and operations team is a long-lasting practice. Both teams have different needs and ideas. The development team would like to release the newest features as soon as possible. On the opposite hand, the operations team would prefer the stability of software over releasing new versions.

The essential idea behind DevOps is quite simple. Build a bridge between development and operations team. The development team should know what needs to be done by the operations team and vice versa. Ideally, operations should be part of the development team, so that everyone has the same knowledge base.

Another approach of building the bridge between those two teams is to merge them, where both developers and operations could do the same work. Develop and deploy their work later on without being dependent on the second team.

With both of those approaches, development may concentrate on creating features to production as fast as possible or delivering on time with good quality instead of blaming the second team for their mistakes. Both approaches support the agile concept of fast development.

Why is DevOps important? In current IT market is dominated by the speed of releasing products. This can be seen by the popularity of agile techniques to shorten development cycles. And when development cycles are fast enough, there is a bigger need to correctly create space where that product can be placed and regularly updated. With DevOps, it is safer to make changes more often because of automated pipelines of the whole deployment [1].

DevOps in practise

DevOps work usually lays in increasing automation and faster deployment process.

First DevOps task is to create an automated deployment mechanism. Deployment strategy is mostly based on deployment scripts or some continuous delivery system, which is triggered by the CI system. Strategies to deploy to different environments such as development or production may differ. While the development environment is usually automated. Deployment to production often needs manual triggering.

Infrastructure as code, provisioning and configuring environments repeatedly and reliably is part of DevOps expertise and can be part of the CI/CD[1] pipeline. Tools such as Terraform, Chef or Puppet are used for this purpose. Infrastructure as code is mentioned in Chapter3

Developers and operators actively monitor applications and services that were developed, both in production or other environments. Monitoring is done for various purposes, such as providing visibility over failures of deployment or quality of provided services. With proper monitoring faster response to bugs and anomalies is achieved which leads to greater customer satisfaction [16].

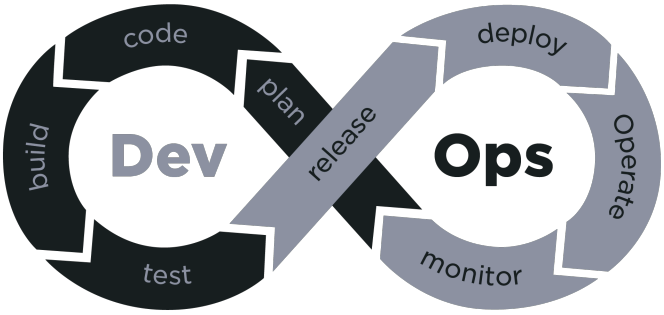

Figure 2.2: DevOps cycle [20]

2.2 Continuous integration and delivery

Continuous integration (CI) and continuous delivery (CD) embody a culture, set of operating principles, and collection of practices that enable application development teams to deliver code changes more frequently and reliably. The implementation is also known as the CI/CD pipeline and is one of the best practices for DevOps teams to implement. [18]

To be able to create your CI/CD pipeline, proper tooling and technology must be chosen. While implementing a CI/CD, teams have to decide which tools fit best in their business and technology stack.

[1] Continuous integration/Continuous delivery

Continuous integration

Continuous integration is a philosophy that supports rapid software development. Operating principles are based on that philosophy and they help to achieve delivering of new code frequently and reliably. Using this method it is easier to detect bugs in code sooner than in large additions of code less often.

Teams that want to implement CI/CD to their business often start with version control systems. Code checking can be done frequently for smaller features but also for longer time frames. Development teams are using different strategies for different cases and define how code is merged into production environments.

There are many techniques like *version-control branching*, which is based on creating a branch for each environment where software is running. One branch is development, for the newest features. The second branch is created for testing, where the testing is done and after all the needed steps are done, code is merged to the production branch which represents the code used in the latest version of the production system.

The second strategy could be *feature flags*. This mechanism is built around turning on or off features at run time. A production system is using master branch code to run. Newest features are flagged and until they are tested, they can not be flagged as production-ready so neither be deployed.

Building the software as a whole is then automated by packaging all the code, database and other components. This packaging may differ depending on which languages are used. [18]

Continuous delivery

Continuous delivery is part of CI/CD that delivers software to its desired environments. Usually, teams have more environments such as development, testing and production. Each of those environments should have same configurations but are for different purposes.

The objective of continuous integration is to gather code at one place to be handed to continuous delivery. After everything is set up, a continuous delivery process could look like this:

First, the code is pulled from a version control system and starts a build of an application. Then the infrastructure as code tool is executed to change required infrastructure in a given environment. This step is more important for a cloud environment as they are more mutable. Next step is moving a built application to the target environment and configuring environment variables dependent on the environment that is being used. After everything is set up, the application is pushed to their appropriate services, such as web servers, API services. Then an application is deployed, the last thing to do is execute any steps required to restart services that are needed for new code to take effect. At the moment when is application successfully deployed, continuous tests are executed, if tests fail rollback will be applied.

More and different steps could be part of continuous delivery. Those which are mentioned here should give a good understanding of a given problematic [18].

Testing in CI/CD

The vast part of CI/CD is testing. The optimal case is to deliver new versions of software as quickly as possible. Also, quality assurance is very important. This means that the CI/CD pipeline should have included various types of tests to be executed in process of delivering

new versions, and in case tests will find an error in code or delivery process, a rescue plan should exist. That rescue plan might be a rollback to the previous version.

However, the best practice in testing is before continuous delivery is executed. Before releasing a new feature, developers should run unit tests, functional tests and regression tests on their local environment. This leads to correct code in version control systems after committing a new portion of code without breaking the working environment.

Testing code is the first part of the testing of the whole software. There are more like performance testing, API testing, security testing, all these can be also automated. The key to automating these tests is the ability to trigger them some easy way such as the command line.

When all testing is automated, it can be integrated into the CI/CD pipeline. Raw code testing can be done in CI while committing or merging with the master branch. Other tests like performance testing could be done only after deploying the new version to the target environment and if those fails, rollback can be executed [18].

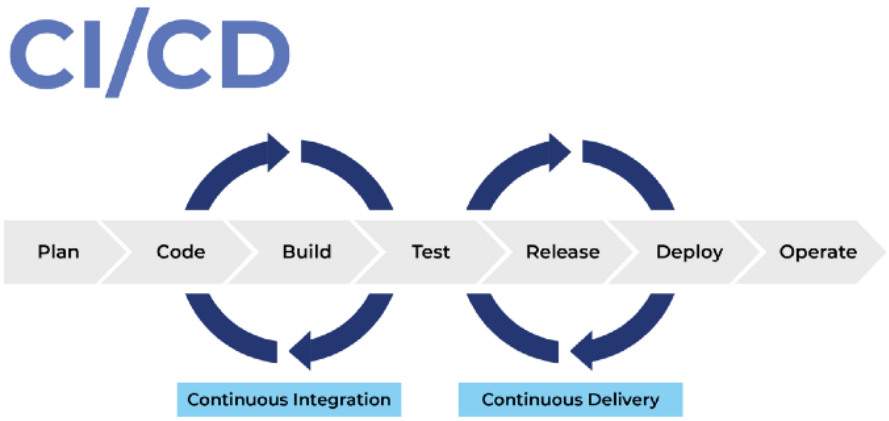

Figure 2.3: CI/CD process [7]

2.3 Team collaboration

In today's world where is a big demand on speed and even more in agile development. Teams have to choose the best way to collaborate. In software development, there are a few points of view. First, that should come in mind is how to effectively share code with the team. In history, before 2005 teams used to share code within version control systems that were centralized and they usually stored each version of the software. They primarily offered prevention of bad things happening, but they did not help in developers daily life very much. Git changed that with its branching system and better control of code [19].

Git

Birth of Git helped developers to create revisions, not only versions of the software. Software development changed because there were many benefits to this approach. Instead of writing a whole new version based on a previous one, teams could easily implement to their workflow small incremental additions. Git offers a branching system, where developers can create

a new branch from the latest version which gives them a complete copy of the software repository and allowing them to do their individual needs such as new feature or bug fix.

Git also keeps a graph which contains a complete history of commits and merging branches. That helps developers to identify problems with each version and can be easily reverted or reviewed.

The big plus of git is that it is decentralized and allows great local development even without internet access. Each individual of a team can clone a repository to a local computer and work with it on their computer. They can commit changes that are ready to be part of the remote repository. Those pushed commits are usually reviewed by other members and then integrated to a master branch which can be production code [19].

GitOps

GitOps uses git repository or another VCS[2] to improve the work of the operations or DevOps team. With GitOps practice configuration files of infrastructure, container-orchestration and other important segments of the software are stored in VCS. Configuration files of infrastructure and other tools are written in a declarative style. These source repositories are becoming a source of truth for the whole project in repositories.

Before GitOps, it was common to write a deployment ticket and wait until an operator successfully deploys the application. Now it is more frequent to edit changes in the repository and create a pull request (PR). After that PR is reviewed by other team members, automated pipeline (CD) is triggered and changes of infrastructure and other configurations are executed.

The fact that GitOps is realized leads to easier testing different environments, reduces „bus factor“, reduces wait time before a new version is deployed and improves overview of infrastructure logic which is handled by infrastructure as code (IaC). Manual toil is also greatly reduced. Very important is that GitOps improves the ability to operate systems safely because operators now do not need to spend so much time with toil[3], they can spend more time on improving CI/CD pipeline which leads to better automation [15].

2.4 Containerization

Containers improve the way the organizations deliver services to end-users. Containers improve agility because applications are a faster and more flexible way than using monolithic architectures which make applications difficult to update. Containers can be shipped as a whole to correct the place and replace the older version of service without noticeable impact. That approach significantly helps to deliver changes sooner than before as it is easier to write the code and create a container [3]. Containers offer light-weight virtualisation, faster than Virtual Machines. Containers provide the ability to manage and migrate application dependencies along with the application with omitting the underlying operating system [8].

The most popular containerization engine is Docker. Docker creates containers with Docker files to create Docker images which are then deployed to the prepared infrastructure manually, or in CI/CD pipeline.

[2] version control system
[3] repetitive time-consuming activity

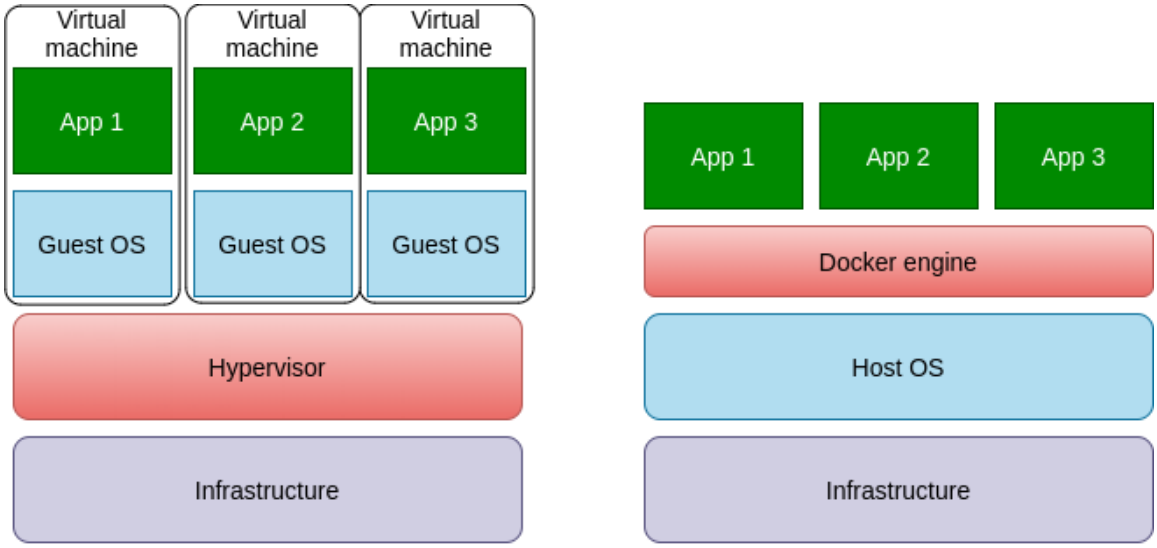

Figure 2.4: Different type of virtualizing application virtual machines versus containerization

2.5 Cloud vs on-premise

Small and medium-sized enterprises (SME) might want to keep their business small or to grow it. When they start to grow it may get harder to manage IT infrastructure. With a bigger company, more on-premise hardware is needed and it can grow gradually or exponentially and takes usually a long time to return on investment [21].

With cloud computing there is no need to take care of your infrastructure, it is provided from a cloud provider in a form of Infrastructure as a service or Platform as a service. Cloud often offers to pay as you go, which means it does not involve large initial investment [8][21].

2.6 Cloud-native

Cloud-native is a well-known term but is not that often described more than „we are on a cloud“. There are many key ideas behind being Cloud-native. One of them is specific design patterns that became very successful while creating cloud applications. Most frequent arguments of cloud-native are as following.

- Cloud-native applications can operate on a global scale. The ordinary web application can be accessed anywhere in the world through the internet. Cloud-native application has replicas of servers and data centres around the whole world so that accessing application results in minimal latencies, for example, google sites can be reached from Europe with lowest latencies, even though Google is located in the United States of America. That is because they have replicas in many places in Europe. This approach creates very robust applications.
- Cloud-native applications have to scale well with many concurrent users. Assumption here that application can horizontally scale automatically. That approach requires careful observation of synchronization and consistency in distributed systems.

- Applications are built on assumption that infrastructure is unstable. Even though one zone of servers will crash down because of some natural disaster the application will still run in a different place so the user does not realize that there is trouble.
- Upgrading or testing Cloud-native applications do not affect end users.
- Security must not be forgotten, cloud-native applications are built of many small components and these components can not hold sensitive data. Access control needs to be managed at multiple levels.

There are many cloud-native applications that the population uses every day but maybe does not know that it is a cloud-native application. For example, The Netflix movie streaming service is one. Also other big players in the current world such as Facebook, Twitter.

At first to become cloud-native, Infrastructure as a Service replaces on-premise infrastructure with virtual machines running in the cloud. It was very difficult to engineer scalability and security at the same time with only on-premise solutions.

The first major design pattern for cloud-native applications was Microservice architecture. This architecture relies on dividing application to small independent components and it easy to scale and reliable. That each component is called microservice. All microservices should be designed for constant failure and recovery.

It must be possible to encapsulate each microservice instance so that it can be easily manipulated. Containerization is the solution.

With all this, it is possible to create a well-developed cloud-native application based on microservice architecture [8].

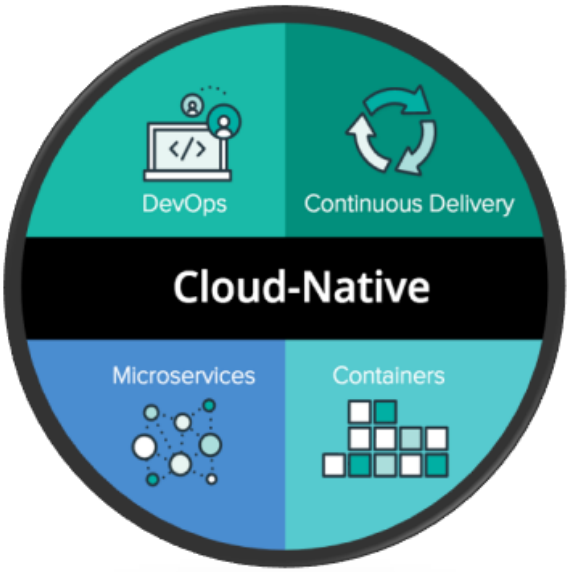

Figure 2.5: Basic principles of cloud-native development [12]

Full stack example of Cloud-Native application

Before creating a new cloud-native application, it is good practice to choose proper tooling, there are a lot of tools for different parts of the application.

On the bottom of the whole application should lay a cloud environment. At the moment the cloud market offers many solutions, most popular are Amazon web services, Microsoft Azure, Google cloud platform, VMware etc.

After the selection of cloud provider, usage of provisioning infrastructure as code tool is desirable to create resources for a project. After choosing the IaC provisioning tool such as Terraform or Cloudformation. Next tool to utilize could be infrastructure configuration tools like Chef, Puppet or Ansible. With these tools, infrastructure is prepared for serving a given purpose.

To be able to deploy to the cloud, developers tend to use runtime environments in which application runs. Those environments are usually created with Container Engines such as Docker, which allows to enclose application with needed components.

Orchestration and Management is the next step of being cloud-native, tools like Kubernetes or Docker Swarm are used to manage container clusters for easy orchestration across multiple hosts. They provide load balancing, scheduling of containers etc.

Many languages support microservice architecture. The ideal one is chosen by the development team. The code is being shipped to runtime services with CI/CD tools such as Jenkins, Travis CI and others.

The last step of cloud-native application is Monitoring, Logging and Auditing. This is one of the key features to manage Cloud Native Infrastructure. All modern monitoring tools support monitoring of containers and microservices [12].

2.7 Cloud Providers

The current market of cloud providing is formed by many professionals. This Section will compare the largest providers which are Amazon Web Services, Microsoft Azure and Google Cloud Platform. Even though these platforms usually provide similar options for enterprise, each has more specific advantages and differ a bit from others. Choice of cloud vendor is on individual customer consideration.

Nowadays, most of the cloud providers offer IaaS[4] and PaaS[5]. Big companies mentioned before are no exception.

Amazon Web Services

At the current state, Amazon Web Services is the largest and maintains 33% share of the worldwide market since they offer a vast amount of services and tools to work with. It offers more than 175 services at a moment across compute, storage, database, analytics, networking, mobile, IoT and more. AWS has been the biggest IaaS provider for over 10 years, it is the most mature provider. However, AWS big weakness is the cost of resources. AWS also focuses on a public cloud rather than a hybrid cloud or private cloud. This implies that sometimes it is not the best choice for an enterprise customer [4].

[4] Infrastructure as a service
[5] Platform as a service

Microsoft Azure

Microsoft Azure is a close competitor to AWS with 18% of market share. Azure is popular due to Microsoft is seen as a safe bet because most clients already have experience with the company. Azure offers exceptional cloud infrastructure and believes that hybrid cloud is important, so it is supportive of private data centres [4].

Google Cloud Platform

Google Cloud Platform is not that big in the competition. Its main benefits compared to other are expertise and industry-leading tools in deep learning and artificial intelligence, machine learning and data analytics. Google has deep expertise around open source technologies. Especially potential lays in containers since Google developed the Kubernetes which is becoming an industry standard [4].

Chapter 3

Infrastructure as a code

Automation of infrastructure is a key DevOps practice. The philosophy behind this is that infrastructure gets a new level of abstraction, infrastructure becomes part of code which describes the desired infrastructure configuration in definition files. That means we can treat infrastructure as another part of the software.

Why is infrastructure as code important? The answer is quite simple. It saves time. It reduces the time spent on doing repetitive things such as patching infrastructure. IaC allowed to create definition files of configuration and so it reduces propensity on errors. Before checklists were made to make sure everything is set up correctly and even though it has a predisposition for human error. Another plus of this that is IaC tools often create snapshots on every version of infrastructure. Because of this, it is simpler to inspect changes with each new infrastructure update.

IaC is improving automation of whole software deployment since infrastructure can be created along with building application. Now there are a lot of tools that can be integrated into CI/CD pipeline such as Ansible, Chef, Puppet etc.

When infrastructure is automated, it is easier to test software in a sandbox environment. It is easy to spin up a new environment, test the new version of the application and then tear down infrastructure in minutes.

In the „cloud age“ it is even easier to run these configurations. Because there is no need to buy physicals servers. It is easier to set up infrastructure on cloud servers which are provided by many companies. On the cloud, there is no need to be afraid of the local server having enough memory because it can be easily added by requesting the provider for more [13].

However, once the infrastructure is managed by IaC tool it might be troublesome to manually debug problems. That is because the IaC tool usually holds a current state which is the last snapshot of applied infrastructure. When is that infrastructure manually changed it may corrupt given state and prevent the tool from working correctly [9].

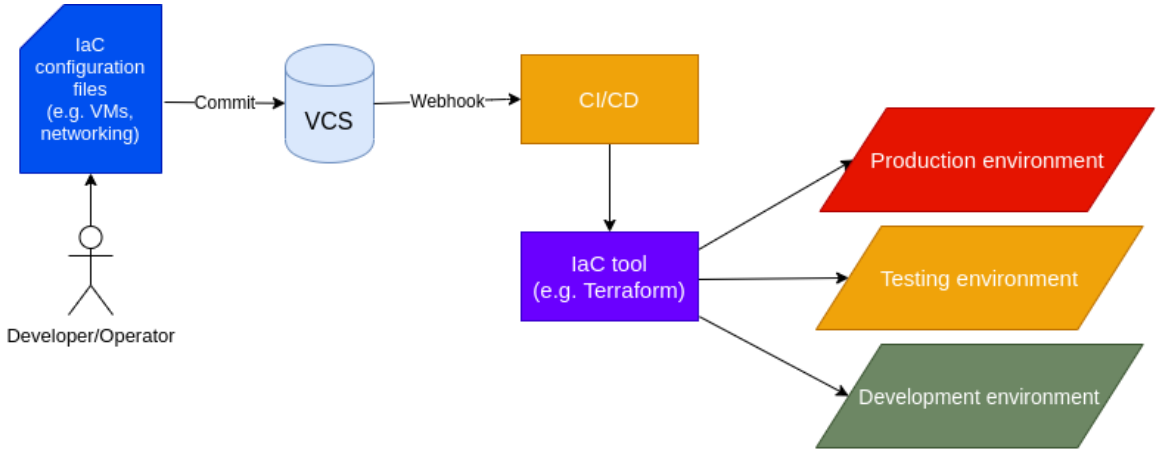

Figure 3.1: IaC is also part of CI/CD pipeline

3.1 Existing Infrastructure as Code Tools

As IaC is rapidly growing many tools are competing at the same time. Different tools are used for different purposes. IaC tools can be divided into four general categories: ad hoc scripts, configuration management tools, server templating tools and server provisioning tools. Where ad hoc scripts are the most straightforward solutions which allow creating infrastructure but without the option of managing it with ease [17][6].

Terraform

Terraform is a server provisioning tool. This means it is responsible for server creation in opposite to IaC tools that install and configure an existing server [6].It is a universal IaC tool that is cloud-agnostic and helps to manage large infrastructure for different kind of applications [5]. Its automation is often different. Some teams run Terraform locally but use the consistent working directory for Terraform to run in. Another approach could use different orchestration tools such as Jenkins to run Terraform [17].

Terraform uses its domain-specific language. Infrastructure is described using high-level configuration syntax. The infrastructure that Terraform can manage includes low-level components such as compute instances, storage to high-level components such as SaaS[1].

The core usage of Terraform is built on planning and applying configuration files. Before creating or updating infrastructure Terraform informs the user about changes in a plan that contains exact information about resource changes in existing infrastructure to increase safety and reduce human error. After inspecting changes, the plan can be applied and infrastructure is updated in a moment [9].

Terraform state

State file which Terraform uses is a mirror of configuration created with Terraform in a text file. Every resource that is stated in this file is currently being managed by Terraform. Unfortunately, the state can not be created by *reading* actual infrastructure.

[1]Software as a service

By default, Terraform state is stored locally in file *terraform.tfstate*. This approach is not ideal while working in a team. To remove this problem, the remote state is an option. The remote state can be stored in a remote data storage. Supported options for storing the state are remote storage buckets, Terraform Cloud and similar options provided by cloud providers. That strategy empowers team collaboration in infrastructure creation [10].

Chef

Chef is a popular IaC configuration management tool among CI/CD practitioners. Chef handles installation and management of software on existing servers. Chef uses Ruby-based DSL to create its *recipies* and *cookbooks*. It has versioning system that allows maintaining a consistent configuration. Each *cookbook* should relate to a single task, but it can deliver different server configurations based on resource definition. Chef uses a procedural approach to its configuration, as describing procedure is necessary to get the desired state.

Thanks to its support for cloud provisioning APIs, Chef works well with other IaC server provisioning tools. Chef is cloud-agnostic and works with many cloud service providers [6][17].

Puppet

Similar to Chef, Puppet is a popular configuration management tool that helps with continuous delivery of software. It has Ruby-based DSL. Puppet which uses a declarative approach to its configuration. Defining the desired state of your infrastructure causes Puppet to automatically enforce the desired state and fixes any incorrect changes. This approach is mainly directed toward system administrators. It can be integrated with the leading cloud providers such as AWS, Azure, Google Cloud and VMware [5][17][6].

Ansible

Ansible is an open-source infrastructure configuration tool. Ansible forms infrastructure by describing relations between system components as opposed to others which manage systems independently. It describes its configurations in YAML in form of Ansible Playbooks, because of that configurations are easy to understand and deploy. Its functionality can be extended by writing new modules and plugins [5].

3.2 Cloud specific IaC tools

A few tools are dedicated to the specific cloud and in general, they provide similar options for users but can not be used for multi-cloud projects as they are integrated into a specific cloud.

AWS CloudFormation

AWS CloudFormation is a similar IaC server provisioning tool to Terraform, but it is deeply integrated into AWS Cloud. AWS CloudFormation does not have its DSL[2] instead it is using JSON or YAML templates or programming languages to describe infrastructure.

This tool allows rollbacking automatically if errors are detected. CloudFormation supports deployment across multiple AWS accounts and regions within a single template [5][17].

[2]Domain specific language

Azure Resource Manager

A similar tool like AWS CloudFormation but this time for Microsofts cloud Azure. Azure Resource Manager creates infrastructure declaratively described in JSON templates. With these templates, it is possible to organize dependent resources into groups which allow to deploy or delete these groups in a single action [5][17].

Google Cloud Deployment Manager

As the title says, an IaC server provisioning tool created specifically for Google Cloud Platform. The tool configuration definition is created in YAML, JINJA2 or Python files.

Google CDM[3] supports previews. This approach allows to examine changes of infrastructure rather than executing changes directly. Human errors can be avoided, which helps to stabilize infrastructure as a whole [5][17].

3.3 Research on similar existing solutions

This Section is focused on exploring existing solutions which are controlling concurrent access to deploying or automating Terraform configuration.

Atlantis

Atlantis is an open-source tool that allows improved collaboration on projects where Terraform is used. Its review and applying system is done in pull requests created in version control system. It allows automation of infrastructure creation. [2]. Atlantis is deployed as a single binary executable. A developer adds a GitHub or GitLab token for a repository containing Terraform code. The installation itself adds hooks to given source repository which allows operating during pull requests. It can run in a container or virtual machine. Requirements are that Atlantis can communicate with VCS and infrastructure that is being changed [14].

Figure 3.2 shows a workflow adopted by developers while using Atlantis. Its workflow is based around feature branches and creating Pull Requests. Communication with Atlantis is defined within comments on Pull Requests, when Pull Request is created, terraform plan is created and Atlantis comments the output of plan back to comment section. Developers then can apply infrastructure by commenting *atlantis apply* on a pull request to apply the planned infrastructure [2][14].

Atlantis is a good tool to add to the CI/CD pipeline while using Terraform. However, it does not support control of concurrent access to infrastructure changes.

Terraform Cloud

Terraform Cloud is a commercial product that offers an extension of Terraform workflow. Working with Terraform Cloud offers its servers to run and maintain Terraform runs on their servers.

With Terraform Cloud practicals it is possible to manage the whole infrastructure automatically or within a nice GUI[4]. Terraform Cloud offers complete control over infrastructure changes, automated planning, applying and storing remote states. Terraform Cloud can

[3] Cloud Deployment Manager
[4] Graphic user interface

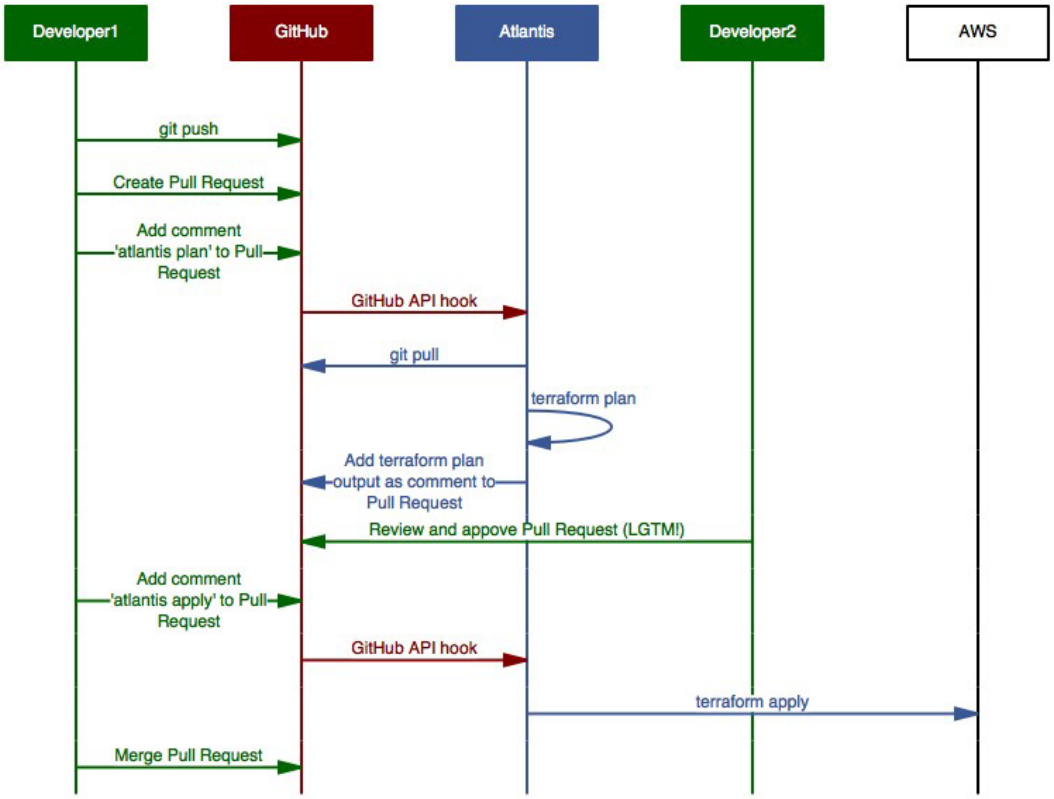

Figure 3.2: Workflow with Atlantis automation tool [14]

be connected with popular VCS such as GitHub or GitLab. It improves team collaboration with nice visible history and well-organized infrastructure operating in a user-friendly GUI [11].

Terraform Cloud comes with improved locking of Terraform state. While all operations run in a central location. It allows detecting attempts of creating a new plan while an existing plan is waiting for approval. These operations can be queued. With this, some control over concurrent accessing to Terraform is implemented [10].

Chapter 4

Design

This Chapter focuses on designing an infrastructure provisioning agent that improves team collaboration and controls concurrent access to infrastructure managing with IaC tool Terraform. The Agent should improve the continuous delivery workflow of Terraform. The workflow between ordinary use of Terraform and using an agent is stated.

4.1 Terraform workflow

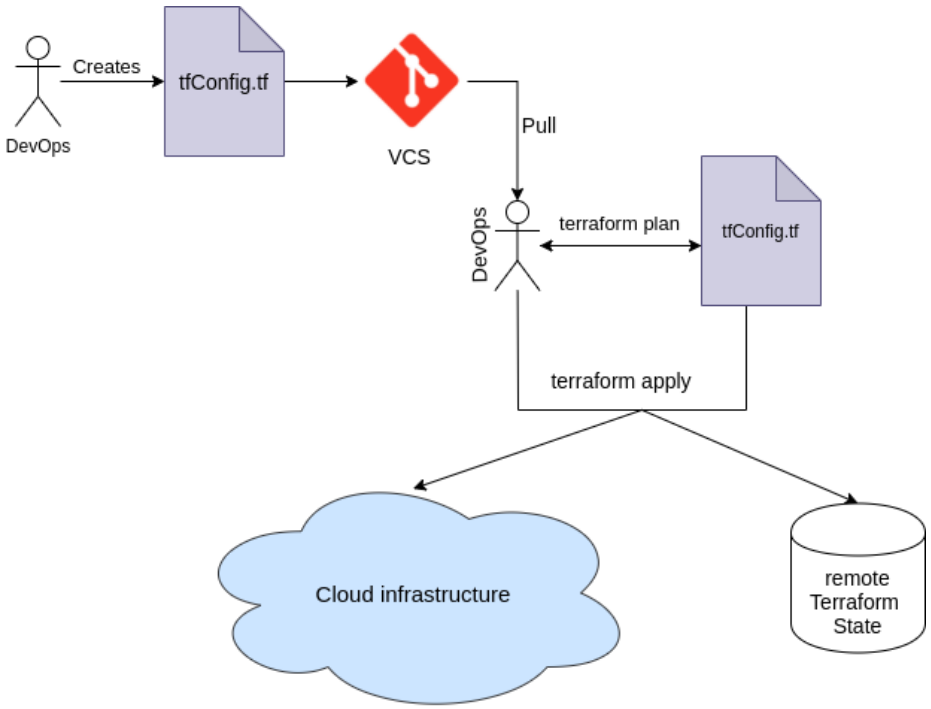

Figure 4.1: Currently adopted style of terraform workflow

Figure 4.1 shows currently recommended basic workflow with Terraform [9]. Its usage is based on sharing Terraform configuration in VCS and having dedicated team members to manage its configuration and infrastructure while having state file in remote storage.

As mentioned in Chapter 3.1. Terraform configuration is based on planning and applying infrastructure with its CLI. Terraform files consist of resources configuration. After running *terraform plan* command it displays required changes. Either it can be creation, updating, replacement or destroying a resource. A user then reviews changes and can run *terraform apply* to apply these changes. All these commands are executed from the Terraform directory on the user computer where Terraform configuration files are located.

Possible issues

Current terraform workflow brings possible issues to flawless development. Especially for those who would like to add terraforming infrastructure to CI/CD pipeline and to larger projects, where may occur multiple concurrent changes at one time. And to who would like to adopt more GitOps style, with version control systems such as git to be the source of truth without need of pulling code to the local computer.

- While working in a larger team on a big project. Each developer may have different requirements for actual running infrastructure. That implies that there may encounter more infrastructure needs and changes. If for example, one developer wants to add one infrastructure resource addition such as database creation and at the same time someone else is upgrading Kubernetes cluster which takes a long time to proceed. An issue can be caused and Terraform automatically locks terraform state from further editing to protect the state from corrupting. This may cause problems such as inconsistent terraform state compared to actual infrastructure and further manual steps are required, such as unlocking state file.

- Second issue is that terraform is being run from a local computer with pulled infrastructure definition files. This approach is not significantly bad but it might cause troubles. For example, a developer which currently want to change infrastructure does not have the necessarily latest version of source repository and applies infrastructure which is not suitable for all team members that are working on the project. The goal is to make sure when somebody decides to update infrastructure, it will be the last possible version which is merged to the master branch.

4.2 Designing the agent

Terraform agent should meet several requirements.

- Agent should allow the concurrent deployment of infrastructure to reduce failures in larger teams or projects.

- Inclusion of Terraform to CI/CD pipeline should be implemented, demand that Terraform should be more GitOps focused should be met.

- Team collaboration improve. Remove the need to have dedicated team members to manage infrastructure.

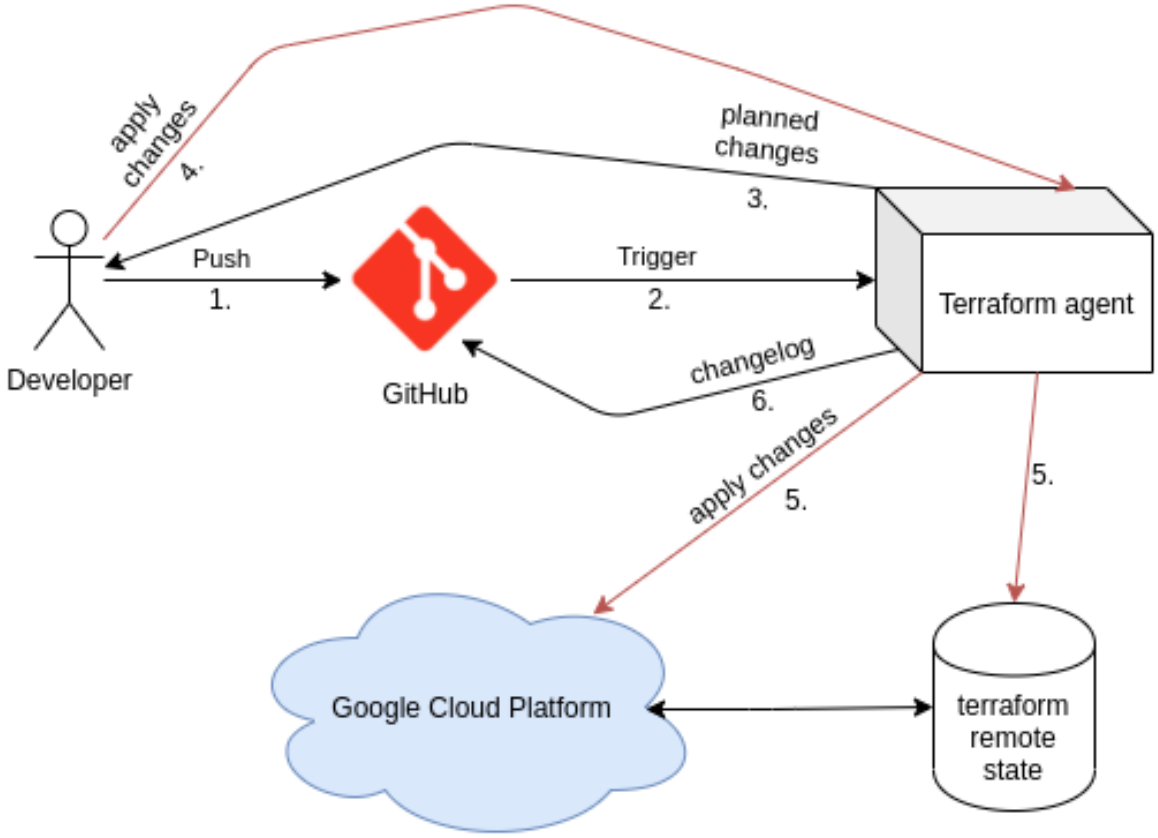

Figure 4.2: Workflow with usage of Agent

Technology specification

Terraform agent is designed to work at Google Cloud Platform, which was chosen because of personal preferences and because the platform offers many important IaaS components that are often used for small and larger projects. Choosing GCP[1] over other cloud providers does not have an impact as the tools used are available at other providers too.

The agent uses for its functionality service called App Engine which is a fully managed service by platform and offers compute power to run applications. App Engine is used for the server part of the agent.

Cloud Build, which is google build-in CI/CD platform is used for running automated triggers to work with GitHub repository.

Server side

The server side of the agent is designed to create plans and apply the newest infrastructure depending on the given source repository. Agent server should be able to use Terraform to plan infrastructure, send planned infrastructure changes and apply those changes after users request.

[1] Google Cloud Platform

Since agent needs requests from a user from client-side or Cloud Build CI/CD pipeline. The proper interface must be chosen. The agent uses the API listener to create responses to requests.

When a successful attempt of infrastructure changes is achieved, the agent writes the last changes of infrastructure back to the source repository. Changelogs should improve visibility of how is infrastructure changed from last time. That information may be valuable for every developer that is involved in the development.

Client side

The client-side is designed to make API calls on the agent server. Main purposes of the client are to review infrastructure changes of actual plan which is sent to client from server and applying that infrastructure plan later on. Another functionality is calling for new terraform plan if something fails on CI/CD pipeline. This should be used only as a backup plan.

API calls are made on a public domain, so some security is necessary. Without security, anyone could run changes on infrastructure without permission.

Terraform agent is designed to improve stated issues in Section 4.1. As shown in Figure 4.2, using agent should remove some manual toil which has to be done by specialized team members. With Terraform agent, developer or operator could add a new configuration and commit it to the source repository. After pushing these changes, an automatical trigger would notice the agent, the agent creates a terraform plan then saves it in local storage. The developer can then inspect changes by calling client-side of the agent to review changes which Terraform planned. When everything seems correct developer can call again the client-side of agent and apply those changes. After applying, the agent will commit changelog of infrastructure to the Github.

Improved team collaboration

Terraform agent is trying to improve the workflow of working with Terraform as a team. In a normal team, a dedicated person which is taking care of infrastructure is needed.

With the agent, that holds credentials to access and manage infrastructure. There is no more need for that person, now everyone can use the tool without having explicit rights to the project by calling from the client application.

The Agent uses the most recent mirror of the source repository. As it is creating plans from latest merged changes. This allows having always a prepared plan for the newest infrastructure.

The important part here is also reviewing changes with the client-side application. That part of infrastructure managing is crucial because interventions to infrastructure should be double-checked, once during pull requests by team members to check if the configuration is correct. The second time when is infrastructure being applied. This approach should significantly reduce the error rate in managing infrastructure and support the smoothness of whole development.

Controling concurrent access

Larger projects may encounter inconvenience while working in bigger teams. Multiple concurrent changes of infrastructure may be one of them. Usually, only one Terraform configuration is created for each project on Google Cloud Platform, that is because if there were multiple terraform configurations it could cause a mess and harder view on

the whole infrastructure. This approach is not best suitable for large projects. Only one configuration means only one terraform state. And when multiple concurrent changes are made, big chance of locking the terraform state comes to play. When is state locked, manual intervention may be necessary.

To prevent these obstacles agent can create a queue of changes, which will prevent Terraform from locking the state. Since the approach of the agent is using actual GitHub repository as a source of truth, not only it does prevent the state from locking by creating a queue. It also uses all merged configurations after last planning to create a new plan with all latest merges to master branch.

Chapter 5

Implementation

This Chapter provides information about the implementation of Terraform agent. The implementation is divided into two code parts which consist of a server and client-side of the application. Essential usage of cloud for correct behaviour of the whole application is also part of the implementation.

Due to requirements described in 4.2 REST API is implemented to answer requests from the client application and the automatic triggers from Cloud Build service.

5.1 Server Side

The server side of the application is designed to create responses to API calling from client-side of application. This Section takes a closer look at the more important parts of the implementation. Application is created in the Go language.

API implementation

After is agent created and successfully deployed into cloud virtual machine, copy of infrastructure repository from GitHub is created. And starts listening for HTTP or HTTPS requests.

API uses the HTTP Echo web framework for its implementation. The Echo responds to three endpoints.

- */terraform/plan*
- */terraform/show*
- */terraform/apply*

Each of these endpoints responses to API calling and runs corresponding routine.

Plan

Calling the */terraform/plan* endpoint results in pulling the latest changes of the source repository, where Terraform configuration files lay. After having the latest configuration, *terraform init* is called via executable of Terraform. *terraform init* initializes working directory and creates the first part of Terraform state within the current working directory. Initialization is important because it prepares Terraform to further callings. It downloads

all provider plugins of providers which are mentioned in Terraform configuration such as Google, Kubernetes, etc. More importantly, it creates a connection to remote storage, where Terraform state will be stored.

After initialization, routine calls *terraform plan* to create a plan of infrastructure changes. Text file with stated changes is stored in virtual machines file system. At the same time, *plan.out* is created. This file is in binary format and contains the execution plan of what needs to be done to create changes in the desired cloud environment and waits in the working directory until *terraform apply* is called.

Show

Calling */terraform/show* results in sending information about planned changes via HTTP response. The response is filled with information about future infrastructure changes which was previously created by *terraform plan*.

This part is essential for developers or operators to be able to preview expected changes. Many unwanted errors may be avoided just by double-checking this report.

Apply

When */terraform/apply* endpoint is hit, routine checks if any plan exists in its working directory and starts executing the Terraform plan created before. Since applying can take a longer time to finish, HTTP streaming is implemented via Gorilla web sockets. First, a handshake is realized, and after that *terraform apply* is called to create resources on the cloud. While execution is in progress, the server sends information about resource creation to the client.

After successful applying of new changes. Log of last changes in file *infrastructure changes* is committed back to the source repository where the configuration is. The execution plan is destroyed, so it can not be reapplied.

GitHub API

GitHub Go library *go-github* is used for communicating with GitHub API where source repository is located. The library is mainly used to communicate via Personal access token.

GitHub access is used to clone and pull repository which is essential for Terraform to run commands. Another usage is for application, so it could create a log of changes in the repository. This approach improves visibility over the whole infrastructure and better tracking of errors.

Containerizing application

The server-side of the application is containerized with Docker. Containerization of the application allows enclosing of necessary components such as Terraform and all used libraries which are essential for the application to run.

Dockering the application is separated into two parts, the first part consists of building the server agent and creating an executable with Go image.

The Second part uses alpine image, to reduce application size at the minimum. Every needed resource is added to make the agent run smoothly. Executable of Terraform is added, so the agent can all Terraform commands inside the container. Also, a service account which has access to Google Cloud Platform project is encapsulated, so the agent

does have the right to change running infrastructure. This service account is mounted as an environment variable, which gives access to the whole container to access specific Google Cloud Platform project.

Controlling concurrent access

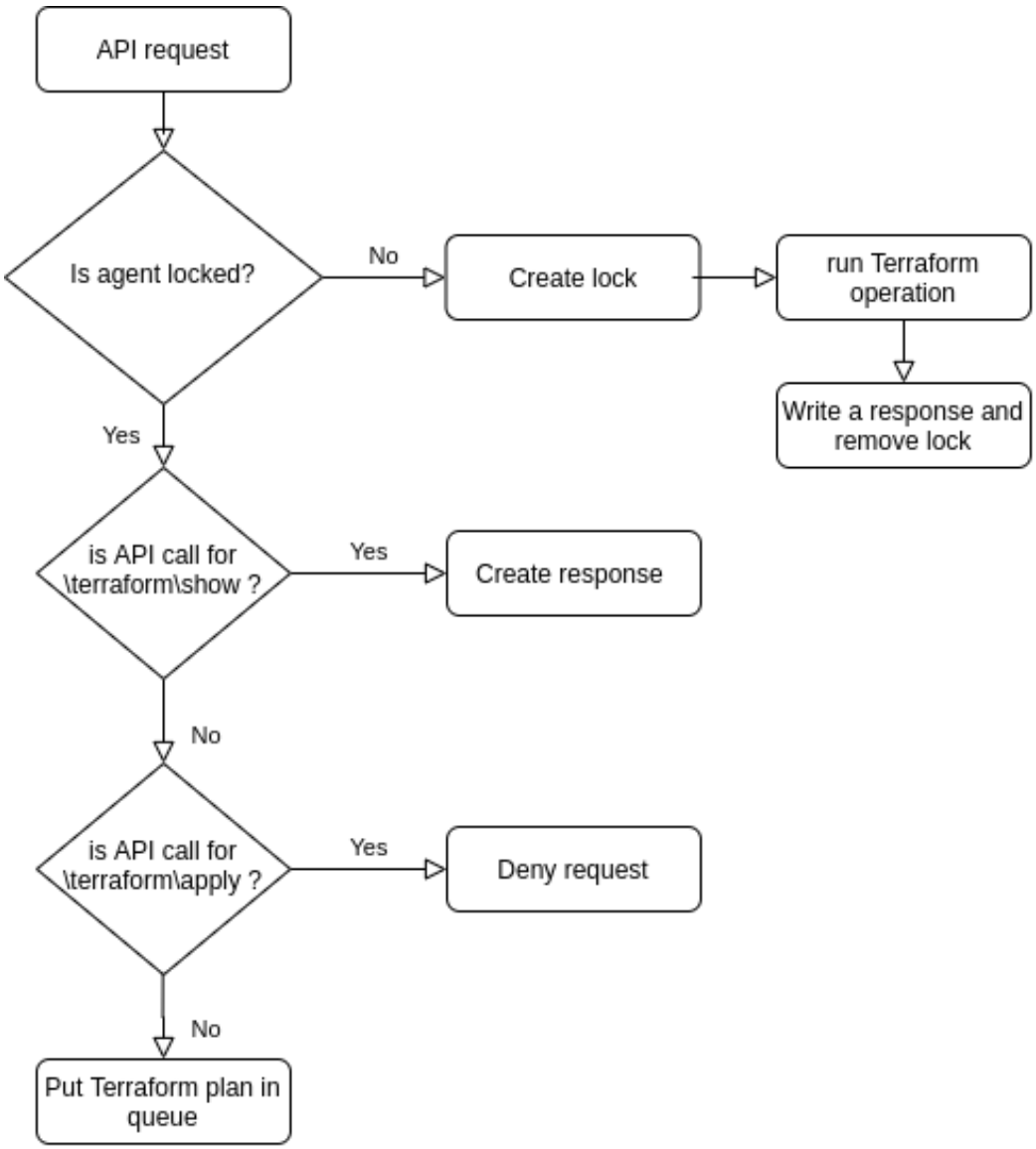

Figure 5.1: Agents decision tree for multiple concurrent accesses

The goal of this work is to implement concurrent access control to infrastructure creation on the cloud environment and solve or evade problems with state locking by controlling it appropriately.

Controlling concurrent access is realized with library *sync*. After *waitgroup* is initialized. All the endpoints share access to that synchronization variable.

When endpoint */terraform/plan* is called, check if there is another running process which is currently working with Terraform remote state is executed. If there is an existing process

such as applying or planning of state at the same moment. Terraform agent prevents to run usual routine after calling an endpoint. Instead, these requests are queued and merged. This principle is derivated from the design itself.

Figure 5.1 shows how the agent responds to each API callings. Hitting */terraform/plan* or */terraform/apply* leads to creating lock which prevents other unwanted callings to be processed.

Terraform plan and Terraform apply are operations that refresh the Terraform state. This means that these two operations can not be called at the same time. In case if */terraform/plan* endpoint is hit twice, it runs a Terraform plan for the first time, second calling is queued and ran after the first one finishes. This has to be done because two separate callings were probably made by an automatic trigger which will be discussed in Section 5.4. And it means that two infrastructure changes have been made. This approach guarantees that the latest configuration of infrastructure will be applied.

When is */terraform/apply* hit, it first checks if *plan.out* exists. This plan is previously created by Terraform plan. If a plan exists, it starts to execute it. However, calling the second time for Terraform apply is prohibited, because applying locks agent from other callings. If */terraform/plan* is hit while applying is in progress, it does queue the request and will be processed later after applying is done.

5.2 Client

The client-side of application is created to call API endpoints of the server. The client is CLI[1] application. This part of the application is very simple.

The client has three commands.Command *plan* which is used to call for planning on the server. This option is here as a backup if automatic planning fails for some reason. *show* calls for information about the current plan and writes text in the standard output. This command should be bread and butter for a developers life as it helps to inspect changes which leads to fewer errors. Both *plan* and *show* receive information in JSON format. On the other hand, *apply* uses streaming for its data transmission as its response may last a longer period.

5.3 Securing connection

Since the agent is divided into client and server and communication is done via open API, some security is required so that everyone can not use that API to change infrastructure on an unfamiliar project.

Each agent is integrated into a project which is an environment for future development. Thanks to that, each agent has a different URL which exposes the API to the internet.

All that is implemented is communicating via an access token, which is transmitted in the HTTP header. The access token is created when the client and server application is built. This approach binds the client and the server without anyone be able to get that token because all the communication is done over HTTPS which is the default for applications that run on App Engine.

[1] command-line interface

5.4 Cloud configuration and deployment

As the agent is designed to work in Google Cloud Platform few configurations must be done for it to work properly. There are two important parts are to deal with. Both are equally important if the agent should work like is described in 4.2.

App Engine deployment

The agent is deployed to App Engine. The application itself is deployed with a file called *app.yaml* and running command *gcloud app deploy app.yaml*. This file contains all the configuration to successful deployment.

```
runtime: custom
service: agent
env: flex
automatic_scaling:
       max_num_instances: 1
resources:
       cpu: 1
       memory_gb: 0.5
       disk_size_gb: 10
```

The configuration is trying to achieve the cheapest configuration for the final virtual machine because it can significantly lower the price.

Custom runtime and the environment is chosen because default options are not viable since the container needs to have encapsulated Terraform binary and other environment settings to work properly.

Trigger configuration

Service Cloud Build is used to automatically trigger routine which hits the *terraform/plan* endpoint of the agent and starts the planning of new infrastructure changes. This plan is then stored in the file system of the virtual machine where the agent runs. That step is crucial for the agent to run as intended because without it it would not be much improvement for the workflow of the Terraform.

Chapter 6

Experiments

This Chapter is containing real-world scenarios where the agent possible could be used and results are stated for each experiment. Both experiments are using GitHub repository called *BP-infratest* and client-side application of the agent called *tfagent*. Both experiments contain just a few resources that are managed by Terraform for easier demonstration. The number of affected resources is not important as the agent should work the same way with less or more of them.

6.1 Experimment 1

The first experiment was designed to demonstrate basic usage of the agent and how it is used in the real world.

After installing the agent to the destination project. Source repository BP-infratest was created on GitHub and connected to Cloud Build service where the trigger is created. That trigger then uses configuration from the *cloudbuild.yaml* located in the GitHub repository.

```
\clodubuild.yaml
\init.tf
\storage.tf
```

Terraform configuration files *init.tf* and *storage.tf* are also created. The *init.tf* contains the configuration of resource providers such as Google, Kubernetes etc. It also has information about backed storage where is Terraform state stored. File *storage.tf* holds configuration of bucket storage. Creation of bucket was chosen just as a demonstration of some non-complex resource which takes short time to create.

Now when the configuration is created only on the local computer, it has to be pushed on GitHub or merged to master via branch merging to run the trigger which starts automatic planning.

Figure 6.1 shows that the automatic trigger is executed and runs proper commands to hit the *terraform/plan* to create the plan. After the trigger is done, the client application runs command *./tfagent show*. The agent responds with infrastructure changes and the user can examine them.

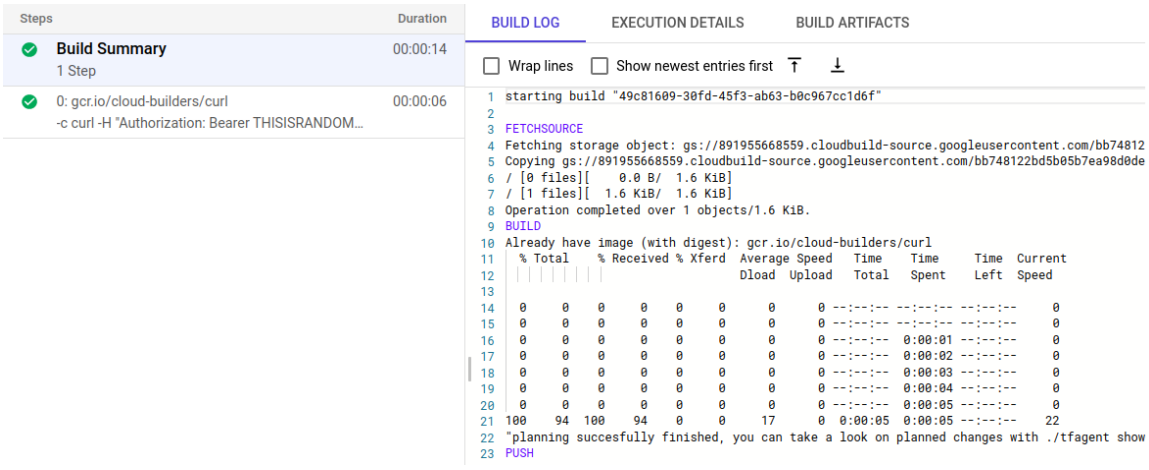

Figure 6.1: Successfully planned infrastructure with trigger

```
Terraform will perform the~following actions:

  # google_storage_bucket.auto-expire will be created
  + resource "google_storage_bucket" "bucket-demonstration" {
      + bucket_policy_only = (known after apply)
      + force_destroy = true
      + id = (known after apply)
      + location = "EU"
      + name = "demonstration-bucket"
      + project = "vojtah-sandbox"
      + self_link = (known after apply)
      + storage_class = "STANDARD"
      + url = (known after apply)
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

When is the user satisfied with stated changes, it can be applied with client command *./tfagent apply*. Which will create real-time communication with server and output is streamed.

```
Applying infrastructure plan
This might take a while...
connecting to server
google_storage_bucket.bucket-demonstration: Creation complete after 1s

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Successfully finished logging on git
```

And at the same time, if the applying of the newest infrastructure is successful. Changelog which has the same format as the output of *./tfagent show* is pushed to GitHub.

The first experiment was successful and did what was expected without any problems.

6.2 Experimment 2

The second experiment is designed to try out the concurrent access control feature. This experiment will test if other requests are processed while Agent will be applying the newest changes.

The expected behaviour is that other requests to apply the infrastructure will be denied as the operation can run applying only once because of shared Terraform state. If a request for planning appears the agent should queue the request and process it after applying is done.

Before creating the second experiment Google Project was cleared from previous experiments. For this experiment, a resource which takes longer time to create is necessary. For demonstration purpose resource which is using sleep is utilised because creating other components such as Kubernetes cluster or database instance are paid and unnecessary for demonstrating functionality.

First will be executed applying of the long creating resource and then experimenting if everything works correctly while creating new requests is done.

After executing the trigger by pushing to the master branch plan is created and then is called *./tfagent show*:

```
Terraform will perform the~following actions:

  # null_resource.sleep will be created
  + resource "null_resource" "sleep" {
      + id = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Then *./tfagent apply* is executed and experimenting can be launched. *null_reource.sleep* creation is configured to last for four minutes which should be enough to demonstrate functionality.

```
Applying infrastructure plan
This might take a while...
connecting to server
null_resource.sleep: Provisioning with 'local-exec'...
null_resource.sleep (local-exec): Executing: ["/bin/sh" "-c" " sleep 240\n"]
null_resource.sleep: Still creating... [10s elapsed]
.
.
null_resource.sleep: Still creating... [230s elapsed]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Successfully finished logging on git
```

While the applying is in run following steps were done:

- Added bucket storage resource in Terraform configuration and pushed to the master branch.
- Added PubSub topic (Google Cloud Platform asynchronous messaging) and merged to maser branch through the pull request.
- Called for *./tfagent apply* for second time.

As the applying of the infrastructure was in the process, the planning was queued up and waited to finish the current operation. After applying was done, both resources were planned automatically and merged into one plan which is wanted behaviour. While trying to call second apply, the client got a response that applying is in progress and exited.

The second experiment can be called success. Concurrent access to infrastructure changes meet requirements stated at beginning of the experiment.

Chapter 7

Conclusion

The world of agile development is based on fast and reliable development cycles. Continuous integration and delivery is a big part of the process because it allows deploying almost instantaneously when a new feature is created. Another big part of agile development is the use of infrastructure as a code. Agile development is often connected to DevOps culture because it builds a bridge between developers and operations teams which provides better team communication starting by developing to deploying the product.

Infrastructure as a code became popular because it allows to abstract infrastructure as another part of the code stored in version control systems. There are many existing tools for managing infrastructure with IaC. This thesis compared popular tools and divided them based on usage for different purposes.

Cloud and cloud-native applications were mentioned as they are increasingly used in modern agile development because they do not need initial capital and also provide a lot of different solutions for various applications.

In the second part a Terraform concurrent agent was designed which works as a Terraform workflow extension. It aimed to improve team collaboration and controlling concurrent access to Terraform state which is held by Terraform and used for its operations.The application is realized as two parts. A server and a client. The server is API which reacts to requests from the client-side of application. The server-side of the agent is running in the Google Cloud Platform and takes advantages of diffrent services provided by the cloud environment. The software is released under MIT license on GitHub. The repository can be found here: https://github.com/hromadkavojta/terraform-concurrent-agent

Two experiments were performed to prove that the agent is working correctly. The first experiment was done to see if the agent is behaving as the proposed design and can improve team collaboration and supports the culture of GitOps. The second experiment had to demonstrate control of concurrent access to Terraform state and that it prevents the state from corrupting or locking. Both experiments successfully used Terraform to create the infrastructure and the second experiment confirmed that the agent controls concurrent access correctly.

However, the price of the virtual machine where the server runs even in the lowest configuration can be a critical aspect of choosing this type of solution over a different solution. This criticism is understandable because monthly it can become an expensive matter in terms of smaller projects. In future development, it would be beneficial to find a solution that does not need to be running constantly to decrease the price of VMs.

Bibliography

[1] Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M. and Tamburri, D. A. DevOps: Introducing Infrastructure-as-Code. IEEE. 2017, p. 497–498.

[2] Atlantis. *What Is Atlantis? TEXu* [online]. [cit. 2020-21-07]. Available at: https://www.runatlantis.io/guide/#getting-started.

[3] Baker, C. *Want to be more agile? Get containers for your apps TEXu* [online]. [cit. 2020-26-07]. Available at: https://www.itworldcanada.com/article/want-to-be-more-agile-get-containers-for-your-apps/421733.

[4] Carey, S. *AWS vs Azure vs Google Cloud: What's the best cloud platform for enterprise? TEXu* [online]. [cit. 2020-15-07]. Available at: https://www.computerworld.com/article/3429365/aws-vs-azure-vs-google-whats-the-best-cloud-platform-for-enterprise.html.

[5] Chan, M. *15 Infrastructure as Code tools you can use to automate your deployments TEXu* [online]. [cit. 2020-15-01]. Available at: https://www.thorntech.com/2018/04/15-infrastructure-as-code-tools/.

[6] Danek, B. *Why Choose Terraform Over Chef, Puppet, Ansible, SaltStack And CloudFormation? TEXu* [online]. [cit. 2020-14-07]. Available at: https://selleo.com/blog/why-choose-terraform-over-chef-puppet-ansible-saltstack-and-cloudformation.

[7] Eneh, T. *Most popular CI/CD pipelines and tools TEXu* [online]. [cit. 2020-10-07]. Available at: https://medium.com/faun/most-popular-ci-cd-pipelines-and-tools-ccfdce42986.

[8] Gannon, D., Barga, R. and Sundaresan, N. Cloud-Native Applications. *IEEE Cloud Computing*. IEEE. 2017, vol. 4, no. 5, p. 16–21. ISSN 2325-6095.

[9] HashiCorp. *Introduction to Terraform TEXu* [online]. [cit. 2020-15-01]. Available at: https://www.terraform.io/intro/index.html.

[10] HashiCorp. *Introduction to Terraform TEXu* [online]. [cit. 2020-18-07]. Available at: https://www.terraform.io/docs/state/index.html.

[11] HashiCorp. *Sign up for Terraform Cloud TEXu* [online]. [cit. 2020-21-07]. Available at: https://learn.hashicorp.com/terraform/cloud-getting-started/signup.

[12] Jog, C. *Cloud Native Applications — The Why, The What The How. TEXu* [online]. [cit. 2020-16-07]. Available at: https://medium.com/velotio-perspectives/cloud-native-applications-the-why-the-what-the-how-9b2d31897496.

[13] Johann, S. Kief Morris on Infrastructure as Code. *IEEE Software*. IEEE. 2017, vol. 34, no. 1, p. 117–120. ISSN 0740-7459.

[14] Kodroff, J. *CI/CD for Infrastructure as Code with Terraform and Atlantis TEXu* [online]. [cit. 2020-22-07]. Available at: https://www.2ndwatch.com/blog/ci-cd-for-infrastructure-as-code-with-terraform-and-atlantis/.

[15] Limoncelli, T. GitOps: a path to more self-service IT. *Communications of the ACM*. ACM. 2018, vol. 61, no. 9, p. 38–42. ISSN 00010782.

[16] Lwakatare, L. E., Kilamo, T., Karvonen, T., Sauvola, T., Heikkilä, V. et al. DevOps in practice: A multiple case study of five companies. Elsevier B.V. 2019, vol. 114, p. 217–230. ISSN 0950-5849.

[17] Nallamala, N. *The Top 7 Infrastructure As Code Tools For Automation TEXu* [online]. [cit. 2020-14-07]. Available at: https://www.ibexlabs.com/top-7-infrastructure-as-code-tools/.

[18] Sacolick, I. What is CI/CD? Continuous integration and continuous delivery explained. *InfoWorld.com*. San Mateo: Infoworld Media Group. 2018.

[19] Spinellis, D. Git. *IEEE Software*. IEEE. 2012, vol. 29, no. 3, p. 100–101. ISSN 0740-7459.

[20] Touch4IT. *DevOps: co to je? TEXu* [online]. [cit. 2020-15-01]. Available at: https://touch4it.cz/blog/devops-co-to-je.

[21] Watson, L. and Mishler, C. From On-Premise Applications to the Cloud. *Strategic Finance*. Montvale: Institute of Management Accountants. 2014, vol. 96, no. 2, p. 80–81. Available at: http://search.proquest.com/docview/1552717174/. ISSN 1524833X.

[22] Younas, M., Jawawi, D. N., Ghani, I., Fries, T. and Kazmi, R. Agile development in the cloud computing environment: A systematic review. *Information and Software Technology*. Elsevier B.V. 2018, vol. 103, p. 142–158. ISSN 1214-0716.

Appendix A

Content of the storage medium

- /bp.pdf is the bachelor's thesis
- /bp/src is the directory containing source files of the thesis
- /agent is the directory containing source files of the application
  - Makefile contains commands to build and deploy the project
  - agent/ folder contains source files of the server-side part
  - client/ folder contains source files of the client-side
  - agent-cloudbuild.yaml configuration for the automated trigger
  - README.md file that contains steps to successfully deploy the agent