

Valtteri Valtanen

IMPROVING THE MAINTAINABILITY AND DEVELOPER EXPERIENCE OF TERRAFORM CODE

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Terhi Kilamo
Vesa Poikajärvi
June 2023

ABSTRACT

Valtteri Valtanen: Improving the maintainability and developer experience of Terraform code
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
June 2023

Infrastructure as code is a popular way to describe and manage cloud infrastructure, allowing developers to manage infrastructure along with all other code. Infrastructure as code is however often neglected, resulting in increasing technical debt and hindrance to maintenance and further development possibilities.

This thesis improves the infrastructure as code implementation of a case project through a constructive research approach. The case project is a production-use multienvironment infrastructure, implemented with Terraform and running on Google Cloud. The shortcomings of the case project were divided into seven action points, each of which was planned and implemented separately. Based on the results of the implementation, a set of best practices for working with infrastructure as code, Terraform and Google Cloud was compiled.

The implementation of the action points was successful, improving the case project significantly. Most important improvement to the case project was the implementation of the stack of modules approach to declaring multiple environments. All in all the implementation shortened the infrastructure as code with some 2400 lines and reduced the need for tedious manual work. In addition, the thesis reaffirmed existing practices and conventions, but did not discover any new significant contributions. The same rules of law apply to infrastructure as code as to all other code. The thesis also verified that making fundamental changes to a production-use infrastructure as code implementation does not pose any blocking issues.

Keywords: infrastructure as code, Terraform, Google Cloud Platform, maintainability, developer experience

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Valtteri Valtanen: Terraform-koodin ylläpidettävyyden ja kehittäjäkokemuksen parantaminen
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Kesäkuu 2023

Infrastruktuuri koodina on suosittu tapa kuvata ja hallinoida infrastruktuuria pilvessä, mahdollistaen infrastruktuurin hallinnan muun koodin ohessa. Infrastruktuuria koodina kuitenkin usein laiminlyödään, johtaen kasvavaan tekniseen velkaan ja vaikeuttaen ylläpitoa ja jatkokehitystä.

Tämä työ parantaa esimerkkiprojektin infrastruktuuri koodina -toteutusta konstruktiivisen tutkimusmenetelmän keinoin. Esimerkkiprojekti on tuotantokäytössä oleva, monen ympäristön infrastruktuuri, joka on toteutettu käyttäen Terraformia ja pyörii Google Cloud -palvelussa. Esimerkkiprojektin ongelmat jaettiin seitsemään toimenpidekohtaan, joista jokainen suunniteltiin ja toteutettiin itsenäisesti. Toteutuksen tulosten perusteella koottiin joukko parhaita käytäntöjä työskentelyyn infrastruktuuri koodina -prosessin, Terraformin ja Google Cloudin kanssa.

Toimenpidekohtien toteutus onnistui, parantaen esimerkkiprojektia huomattavasti. Tärkein parannus esimerkkiprojektiin oli ympäristöjen määrittelyssä käytettävä moduulipino-malli. Kaiken kaikkiaan toteutus lyhensi infrastruktuuria koodina noin 2400 rivillä ja vähensi tarvetta pitkäkeiselle käsin tehtävälle työlle. Lisäksi työ vahvisti olemassa olevia käytäntöjä, mutta ei löytänyt uusia merkittäviä kontribuutioita. Samat ohjelmoinnin perussäännöt koskevat infrastruktuuria koodina kuin kaikkea muutakin koodia. Työ myös vahvisti, että tuotantokäytössä olevan infrastruktuuri koodina -toteutuksen muokkaamisessa ei ole pysäyttäviä esteitä.

Avainsanat: infrastruktuuri koodina, Terraform, Google Cloud Platform, ylläpidettävyys, kehittäjäkokemus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

After six years, my time at Tampere University has finally reached its end. It has been a blast, thanks to a great group of friends, providing infinite laughs, support and peer pressure. I also want to thank the people making this thesis possible.

I want to thank the product owner and organization of the case project for flexibility concerning my billable work and allowing me to utilize our project in this thesis. Working in the project has been a joy, and it was a blessing to be able to utilize our work in the thesis. I also want to thank my employer, Futurice, for allowing me to work fewer hours in order to complete the thesis. Finally, I want to thank my examiners Terhi Kilamo and Vesa Poikajärvi for their guidance and feedback during the thesis process.

Tampere, 20th June 2023

Valtteri Valtanen

CONTENTS

1.	Introduction	1
2.	Processes and tools	2
2.1	DevOps	2
2.2	Infrastructure as code	3
2.3	GitOps	5
2.4	Tools	6
2.4.1	Terraform	6
2.4.2	Google Cloud	8
3.	Case project	9
3.1	The project and its current infrastructure	9
3.2	Problems with the current implementation	11
3.2.1	Managing secrets	12
3.2.2	Using Cloud Build for CI/CD	13
3.2.3	Module usage	14
3.2.4	Declaring new resources and managing environments	16
3.2.5	Miscellaneous	17
4.	Motivation and research questions	19
5.	Implementation plan	21
6.	Implementation	23
6.1	Action point 1: Refactor modules to be more universal	23
6.1.1	Pub/Sub modules	23
6.1.2	Cloud Scheduler modules	24
6.1.3	Cloud Build trigger modules	26
6.2	Action point 2: Stack of modules	27
6.3	Action point 3: Remove all instances of Terraform secret value generation	30
6.4	Action point 4: Develop a universal deployment trigger	32
6.5	Action point 5: Version Terraform and providers properly	34
6.6	Action point 6: Tighten the IAM roles around invoking Cloud Runs	35
6.7	Action point 7: Cloud Run instances referencing themselves	36
7.	Results and evaluation	38
7.1	Evaluation of the implementation	38
7.2	Answers to the research questions	40
7.3	Best practices	40
8.	Conclusion	43

References. 45

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
CI/CD	Continuous Integration / Continuous Delivery
GCP	Google Cloud Platform
HCL	HashiCorp Configuration Language
HTTP	Hyper Text Transfer Protocol
IaC	Infrastructure as Code
IAM	Identity and Access Management
IT	Information Technology
Monorepo	single repository, a software-development strategy in which the code for multiple applications or projects are stored in the same repository
npm	Package manager for JavaScript
psql	PostgreSQL interactive terminal
URL	Universal Resource Locator

1. INTRODUCTION

Infrastructure as code, or IaC, is a popular way to describe and manage cloud infrastructure. It allows developers to manage infrastructure in the same way as they manage all other code. However, infrastructure as code is often neglected and not given the same attention and care as other code, but used as a one-off script to create the infrastructure. This leads to technical debt, which in turn makes further development and maintenance of the infrastructure more difficult.

This has already occurred in the case project of this thesis. The infrastructure as code implementation has multiple shortcomings, which hinder the maintainability and developer experience of the infrastructure. The goal of this thesis is to improve the maintainability and developer experience of the case project infrastructure as code implementation, allowing for easier maintenance of the infrastructure and enabling more development options. This thesis aims to understand how to improve the maintainability and developer experience of an infrastructure as code implementation and how to make fundamental changes to the IaC when already in production, through the process of improving the case project. In addition, based on the results a set of best practices for working with infrastructure as code and Terraform are compiled.

The thesis is conducted as a constructive research, a methodology used for solving real life problems and thus producing contributions to the field of science where it is applied [38, 40]. In this thesis, the case project is introduced, shortcomings examined, and improvements are planned and implemented. The results are then evaluated, and the best practices compiled.

Chapter 2 introduces the tools, processes and core concepts related to this thesis, starting from DevOps and infrastructure as code to Terraform and Google Cloud. Chapter 3 introduces the case project along with its shortcomings. Chapter 4 discusses the motivation for this thesis and defines the research questions along with the research methodology. Chapter 5 presents the plan in contrast to the problems in the current implementation. Chapter 6 describes the implementation process and the changes made to the infrastructure as code. Chapter 7 presents and evaluates the results of the thesis, and presents the set of best practices. Chapter 8 concludes the thesis.

2. PROCESSES AND TOOLS

This chapter introduces the key processes and tools related to this thesis. The chapter introduces DevOps, infrastructure as code, GitOps, Terraform and Google Cloud Platform on a sufficient level for understanding this thesis.

2.1 DevOps

DevOps is a collection of practices, tools and cultural philosophies that combine the disciplines of development (dev) and operations (ops). DevOps accelerates organization's ability to deliver applications and services through automation, collaboration, continuous feedback and iterative development. In a DevOps team developers and IT operations work together collaboratively throughout the application lifecycle. In DevOps, development and operations teams are no longer separate, and may even be completely merged together. DevOps teams use tools and practices to reliably automate and accelerate processes, and operate and evolve applications quickly. These tools also allow the DevOps teams to handle all kinds of tasks, such as provisioning infrastructure, without the help from other teams. [3, 19, 69, 70, 73]

The DevOps lifecycle is a loop, as presented in Figure 2.1. The loop consists of multiple steps: planning, implementing, testing and verifying, deploying, operating, monitoring and receiving feedback after which it starts again from planning. The lifecycle is short, meaning changes are small and many. [22] DevOps integrates new code continuously, always keeping it in a deployable state [6].

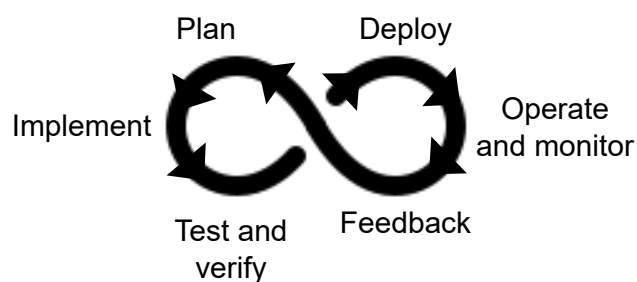


Figure 2.1. DevOps lifecycle, modified from [19, 22, 69]

In addition to the DevOps lifecycle, DevOps has four core principles. They are automation of the software development lifecycle, collaboration and communication, continuous improvements and minimization of waste and hyperfocus on user needs with short feedback loops. [69] The most important of these principles for this thesis is the automation of the software development lifecycle. Continuous integration and continuous delivery are key parts of this automation, and the foundation of DevOps.

Continuous integration (CI) is the automation of code changes into the project. In continuous integration developers frequently merge code changes into a remote repository, where the code is built and tested. This enables the DevOps teams validate the code quicker, address bugs and improve software quality. The code in the remote repository stays at a deployable state at all times. Continuous deployment (CD) extends on the idea of continuous integration, by automatically deploying the code changes to an environment. CI/CD is achieved with an automated pipeline, handling all the steps from integrating the code to building, testing, and finally deploying the code into use. [1, 19]

2.2 Infrastructure as code

Infrastructure as code, or IaC, is an IT infrastructure management process that applies DevOps practices introduced in Section 2.1 to the management of infrastructure [7, 20]. The idea of infrastructure as code is that developers write and execute code to define, deploy, update, and destroy infrastructure [6, 36]. Almost everything can be managed with code, including but not limited to servers, databases, application configuration, networks and log files [6, 7].

Infrastructure as code codifies infrastructure resources into text files, allowing them to be committed into a version control system like Git. When in version control, the infrastructure as code can be treated as all other code, utilizing feature branching and being a subject to code reviews to name a few. In short, version control enables applying DevOps practices to infrastructure as code. A DevOps-enabled infrastructure as code can also utilize automatic deployments and rollbacks in case something goes wrong. [19, 36]

Infrastructure as code also has its own principles: reproducibility, idempotency, composability, and evolvability. Reproducibility means that the same configuration can be used to reproduce the infrastructure resources. Adding infrastructure as code to version control and keeping it up-to-date helps to conform to reproducibility, as the same version of configuration is available for everyone. Idempotency means that the same configuration can be run repeatedly, without any effect on the end state of the infrastructure or undesired side effects. Idempotency ensures that when for example an infrastructure as code that defines two servers is run multiple times, the end result is still two servers. Composability ensures that any combination of infrastructure resources can be assembled and updated without affecting the others. Lastly, evolvability ensures that the infrastructure can be

scaled with minimal effort and risk of failure. [67]

Infrastructure as code tools can be divided into five categories, as presented in [6]: scripts, configuration management tools, server templating tools, orchestration tools and provisioning tools. Scripts are essentially automating anything the developer would otherwise do manually. It is the most straight forward way to implement infrastructure as code, as the developer can write them in any programming language. However, scripts are not very maintainable, and can quickly develop into a mess because all tasks require custom code.

Configuration management tools, such as Ansible, install and manage software on existing servers. In comparison to scripts, configuration management tools adhere to idempotency principle. Scripts are often run just once, as developing idempotent scripts is hard. Using configuration management tools not only bring idempotency into the equation, but also make following conventions and distributing the infrastructure as code easier. [6]

Server templating tools, such as Docker and Packer, are an alternative to configuration management tools. Instead of managing software on existing servers, server templating tools create an image of a server, containing snapshot of the entire server, including the operating system. There are two broad categories of images: virtual machines and containers. Virtual machines emulate the entire system, including hardware. Packer is used to create virtual machine images. Containers emulate the user space of an operating system, which is running on a container engine on the host system. Docker is used for creating and running container images. The case project, presented in Chapter 3, utilizes Docker images for running the applications on Google Cloud. Server templating is an essential step towards immutable infrastructure, as changes to the servers require creating a new image of the server instead of directly modifying the existing server. When the servers never change, it is a lot easier to track what is currently deployed. Orchestration tools, such as Kubernetes, can be used to manage multiple containers created with server templating tools, handling tasks such as monitoring, updating, and scaling the containers and routing traffic. [6]

Provisioning tools, such as Terraform, are used to create the underlying servers themselves, whereas the above-mentioned tools define the code that runs on each server. In addition to creating servers, provisioning tools are used to create networks, databases, queues and almost everything else in an IT infrastructure. Provisioning tools, most notably Terraform, are commonly used to control infrastructure running on a cloud provider's servers. [6, 36]

Although the tools overlap in functionality, in practice multiple tools need to be combined, as each have their own strengths and weaknesses. When working with virtual machines, a common combination is Terraform and Ansible, where Terraform is used to provision the virtual machines and Ansible controls the software on them. [6] Another popular

combination is Terraform and Docker, where Terraform creates managed resources on a cloud provider and Docker builds images to be run on the cloud. The case project also uses Terraform and Docker as infrastructure as code tools.

2.3 GitOps

GitOps is a strategy for implementing continuous deployment for cloud native applications, introduced by Weaveworks in 2017. GitOps is a developer-first approach to operating infrastructure, which utilizes familiar tools such as Git, infrastructure as code and continuous integration. The idea behind GitOps is having a Git repository containing declarative descriptions of infrastructure and an automated process to make the environment match the declared state in the repository. With GitOps, managing the infrastructure is done through the repository, which acts as the source of truth. Git also acts as a history and an audit log, as all changes are committed to the repository through peer reviewed pull requests. In case of failure in an environment, the responsible change can be identified using Git and a rollback issued by simply using `git revert` on the faulty commit. [4, 53]

There are two different ways to implement the deployment strategy for GitOps: push and pull deployments. Push deployments are implemented by CI/CD pipelines briefly introduced in Section 2.1. The pipelines are only triggered by changes made to the repository. When the code in the application repository is updated, the build pipeline builds new images and updates the environment repository. The deployment pipeline then reacts to these changes and updates the actual environment. [4] The push-based deployment strategy is illustrated in Figure 2.2.

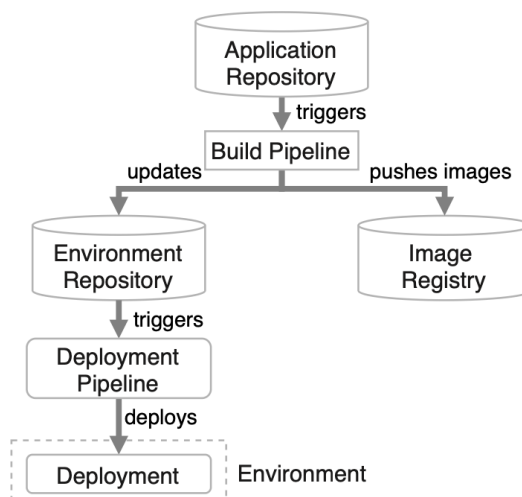


Figure 2.2. Push-based deployments [4].

The other strategy is a pull-based deployment strategy, illustrated in Figure 2.3. The build pipeline in pull-based deployment works similarly to the push-based deployment,

but whereas the deployment pipelines of push-based deployment react to an event, pull-based deployment introduces an operator. The operator controls the deployment pipeline by continuously comparing the desired state of the infrastructure defined in the repository to the actual deployed state, and updates the environment accordingly. In addition to functioning as the push-based deployment, operator also notices changes in the other direction, allowing changes made directly to the environment to be rolled back. [4]

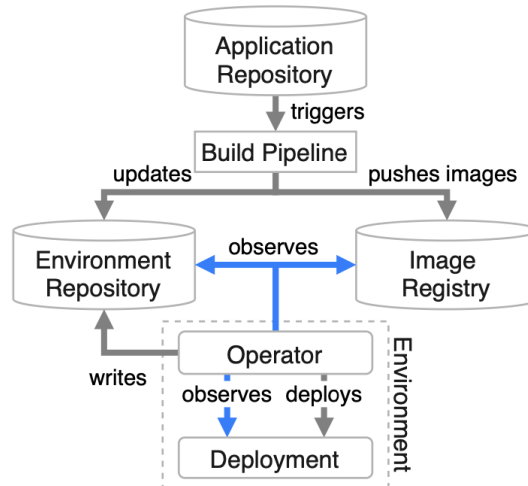


Figure 2.3. Pull-based deployments [4].

One key difference between the strategies is the location of the operator. The deployment pipeline in push-based strategy is external to the environment, requiring credentials to the environment be given to the pipeline, essentially creating a god-mode pipeline. The operator on the other hand lives inside the environment, so no external entity needs credentials to the environment as the changes are made within the environment itself. [4]

2.4 Tools

This section introduces the core tools used in the case project, Terraform and Google Cloud. Terraform is used as the infrastructure as code provisioning tool, to manage the infrastructure on Google Cloud, a cloud platform by Google.

2.4.1 Terraform

Terraform is a cloud-agnostic, open-source infrastructure as code tool first introduced in 2014, that enables defining both cloud and on-premise infrastructure in a human-readable format. [63, 72]. Terraform files can be versioned, reused and shared. It allows for the entire infrastructure to be provisioned and managed via a consistent workflow. Terraform can be used to manage both low-level components such as computing, storage and network resources and high-level components such as software-as-a-service resources. [72]

In 2021, Terraform had been downloaded for 100 million times, and had over a thousand providers and 5 500 modules [63].

Terraform is built on plugins called providers to interact with cloud provider APIs. Terraform configurations must declare the required providers. The providers add resources and data sources for Terraform to manage. Most providers are for specific infrastructure platforms, such as Google Cloud, but providers can also offer utilities, like random number generation. The main directory for providers is Terraform Registry. [46]

The Terraform workflow is divided into three steps: write, plan and apply. Writing consists of creating the Terraform configuration or code, which is what this thesis also focuses on. The code is written using Terraform configuration language, using a syntax called HCL or HashiCorp Configuration Language. When writing the code, `terraform validate` can be used to check that the configuration is valid. The plan step consists of using `terraform plan`, which previews the changes against the existing infrastructure. The plan step can be also used for checking for errors during the writing of the configuration. Finally, apply step executes the planned changes to the real infrastructure. The workflow is a loop, starting from the beginning when more changes are needed. [61]

Resource blocks are the most important entity in Terraform. Each resource block describes one or more infrastructure objects, such as computing instances. [52] When applying a Terraform configuration, one of four things will happen regarding resources: Terraform will simply create or destroy the resources, per the configuration. If the resources arguments have changed, Terraform tries to update the resource in-place. If updating in-place is not possible due to providers API limitations, Terraform will destroy and recreate the resources. [50] Data sources are a special kind of resource, which allow Terraform to use information defined outside the Terraform configuration [17].

Multiple resources can be bundled into modules, which can be reused. All Terraform configurations contain at least one module, the root module. The root module can call child modules to include their resources into the configuration. Modules can be used locally, or published to Terraform Registry. [44]

Resources and modules support meta-arguments, such as `for_each`, `depends_on` and `lifecycle`, which allow for more fine-grained control over the sources and enable implementing more complex use cases. [44, 52] In addition to meta-arguments, Terraform also supports a wide range of expressions. The simplest expressions are just literal values, but there are also more complex expressions like references, arithmetic and conditional evaluation. Other types of expressions include operators, splat expressions, dynamic blocks and version constraints to name a few. [25]

2.4.2 Google Cloud

Google Cloud, or Google Cloud Platform (GCP), is a cloud platform provided by Google, first introduced in 2008, when Google unveiled the App Engine. Through Google Cloud Google offers the same infrastructure it uses internally for its own products. [32] Google Cloud is available from across the world, divided into regions and zones. Regions are independent geographic areas that are further divided into zones. A zone is a deployment area for Google Cloud resources within a region. [27] The platform offers multitude of services, ranging from managed software-as-a-service, solutions to infrastructure-as-a-service. For example, on the computing side developers can choose between fully-managed serverless Cloud Functions, platform-as-a-service solution Cloud Run running containers or self-managed virtual machines via Compute Engine. [32]

The computing resource utilized in the case project is Cloud Run. Cloud Run is a managed computing platform running containers. The containers are run directly on Google's scalable infrastructure. Cloud Run can run programs written in any programming language, as long as they are packaged into a container. In addition, for the most popular platforms like Node.js, Go and .NET Core, Cloud Run offers building the containers for you, straight from the source code. Cloud Run offers two ways to run the code: services and jobs. Services respond to requests, whereas jobs are invoked to run a specific task to completion. Cloud Run also offers integrations to the Google Cloud ecosystem, allowing to build full-fledged systems. [68]

Some other utilized resources are Cloud Storage, Cloud SQL, Pub/Sub and Cloud Scheduler. Cloud Storage is a managed resource for storing unstructured data, commonly used for backups and archives, media content storage and delivery, and serving static websites [12]. Cloud SQL is fully managed relational database service, supporting MySQL, PostgreSQL and SQL Server. Cloud SQL offers integrations to all types of computing available on Google Cloud, effortless scalability, top-tier availability and compliance with a wide range of standards. [11] Pub/Sub is a managed, asynchronous and scalable messaging service. Pub/Sub requires systems of event producers and consumers, called publishers and subscribers. Pub/Sub enables publishers to communicate with subscribers asynchronously, decoupling the systems. Pub/Sub is commonly used for ingesting user and server events, enabling parallel processing and load balancing for reliability. [71] Cloud Scheduler is a managed cron job service, allowing to schedule virtually any job. Cloud Scheduler provides guaranteed at-least-once delivery, configurable retrying policies and multitude of supported targets. [10]

3. CASE PROJECT

This chapter introduces the infrastructure under development at its current state and inspects the implementation choices and reasons behind them. The chapter will also present the challenges that have risen from the choices made during a year of development, leading to the research question and the goals of this thesis presented in Chapter 4. Note that the presented code examples might not be complete and working as is, as they may have been truncated for brevity.

3.1 The project and its current infrastructure

The system at hand is an online payment gateway backend implemented using TypeScript and Node.js, backed with PostgreSQL databases. The current infrastructure for the system is on Google Cloud. The infrastructure is implemented utilizing managed services as much as possible, such as Google Cloud SQL for database needs and Google Pub/Sub for message brokering. Computing is handled using Google Cloud Run instances, as Cloud Functions were not available on the desired region at the start of the project. The application also utilizes multiple schedulers, private networking, load balancing and storage buckets, all provided by Google as managed services.

The infrastructure for the system is provisioned using Terraform to three different environments, test, staging and production, on Google Cloud. These environments are separated from each other by creating separate Google Cloud projects for each of the environments, as suggested in [51]. The infrastructure as code implementations for these environments are completely separate, although utilizing shared Terraform modules to some extent. The Terraform code is executed in the environment which the code concerns, using Cloud Build triggers. These triggers fetch the Terraform code from the correct branch in the project Git repository. Git branch `develop` corresponds with test environment, `release` with staging environment and `main` with production environment. The test environment is used for developing the system, automatically deploying all changes after merging the changes to `develop`. The staging environment is used for testing release candidates, before the final release is deployed into the production environment. Staging and production environments require manual deployments.

The infrastructure implementation follows the GitOps paradigm introduced in Section 2.3,

based mainly on best practices by Google as outlined in [5, 42]. The deployment strategy is a hybrid between the push- and pull-based deployments introduced in Section 2.3. The deployments are triggered mostly according to the push-based deployment. However, in this project the deployment pipeline does not require all powerful credentials to the environment as is usually necessary in push-based strategy. As mentioned earlier, the infrastructure changes in this project are executed using triggers inside the target environment, similarly to the operator in pull-based strategy. However, the deployment of application images is managed by a separate Google Cloud project, requiring some credentials to be granted. The initial plans for the infrastructure are shown in Figure 3.1, highlighting all the projects on Google Cloud Platform and main idea of how application images are deployed to the different environments.

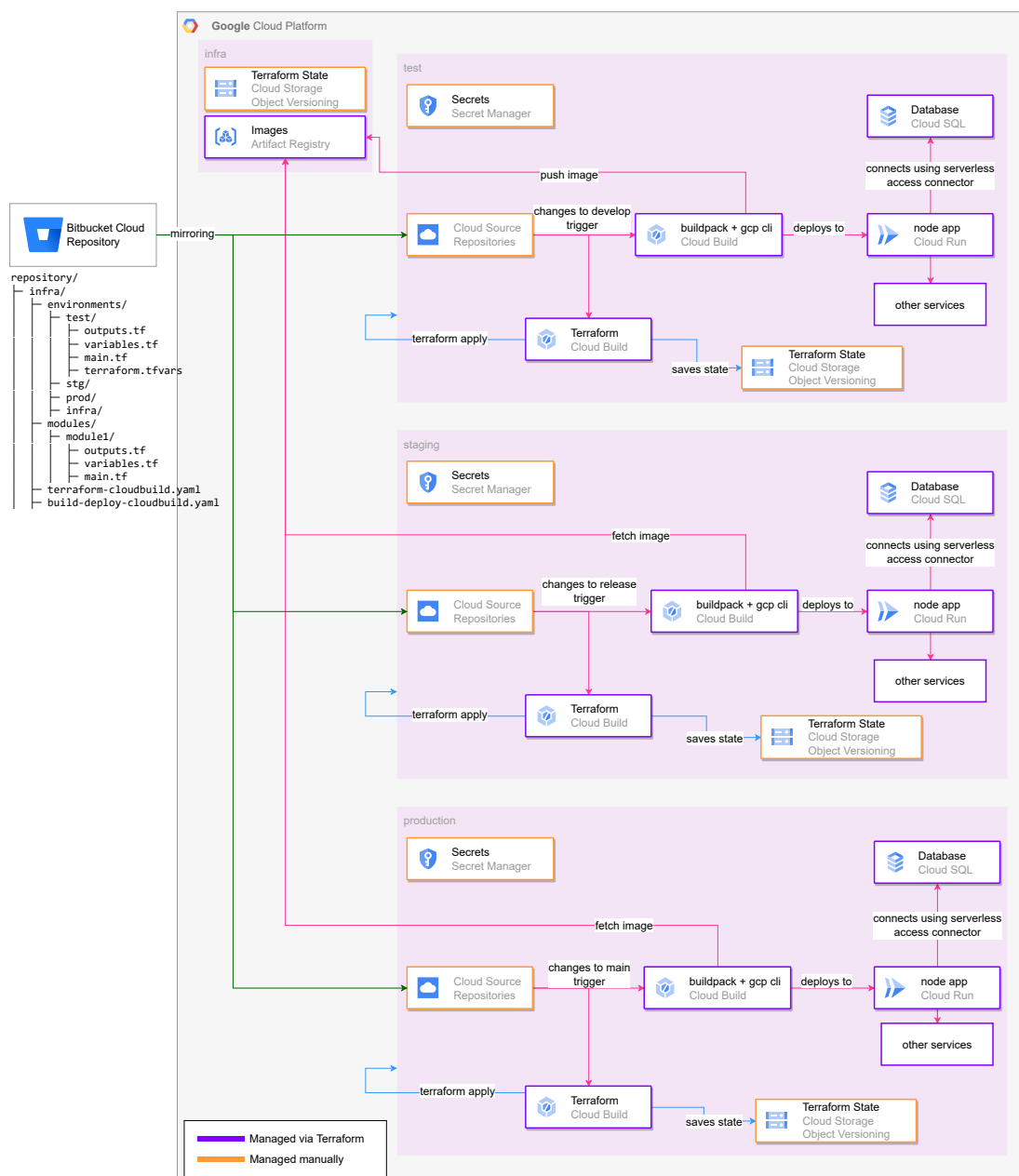


Figure 3.1. Infrastructure diagram of initial plans, June 2022.

In addition to the three application environments presented in Figure 3.1, the infrastructure also includes a meta project called `infra`, which includes resources used by all the other environments. The `infra` project contains Google Artifact Registry for storing application images. Furthermore, the `infra` has evolved to include the Google Cloud Build triggers for deploying said images as well as running database migrations for each of the environments, which were not envisioned during the drawing of the diagram. The project also includes a `seed` Terraform code. The `seed` Terraform is used to initialize the environments presented in Figure 3.1. The `seed` Terraform code enables needed Google Cloud products and sets correct privileges for the Cloud Builds executing the Terraform code and creating Google Cloud resources.

The directory structure of the project, from the parts that are relevant for this thesis, is divided into three folders: `modules`, `environments` and `seed`. The `modules` folder includes currently about 15 self-built Terraform modules. Some modules are quite universal, such as `cloud-run-api`, which is used for creating all Cloud Run instances for running containers. Others are very specific, for example `database` which includes database-related resources, such as database instance and database users with their passwords. These modules are then used in each of the environments with appropriate configurations. The current module usage is further examined in Subsection 3.2.3. The `environments` folder holds the declarations for each of the environments, or GCP projects, which are `test`, `staging`, `production` and `infra`.

The infrastructure has evolved during a year of development, but the core idea presented in Figure 3.1 still remains. Currently, the source code lives in GitHub, and is connected directly to Cloud Build [16] without the extra step of first cloning the repository to Cloud Source Repositories, as is shown in Figure 3.1. Other notable difference when comparing the current infrastructure to Figure 3.1 is that Cloud Native Buildpacks [29] are not used. The building of the applications is handled with self-declared Dockerfiles and a Docker cloud-builder. Main reason against the use of buildpacks was the uncertainty of how they would work with NPM monorepos, when the development team had previous experience with declaring Dockerfiles for monorepos.

Overall the infrastructure is in a fairly good state, but it has some shortcomings which should be addressed in order to ensure proper maintainability and allow for more efficient further development. These shortcomings will be examined in Section 3.2.

3.2 Problems with the current implementation

As mentioned in Section 3.1, the current IaC implementation contains some points of difficulty which will be more closely examined in this section. The major difficulties are presented with examples of how they affect the maintainability and further development of the IaC solution and possible actions what could be done about them.

3.2.1 Managing secrets

The secret management is based on the usage of Google Cloud Secret Manager. The secrets are created by Terraform modules where the secret is needed, which in itself poses no problems. However, there is no unified way of managing values of these secrets within the Secret Manager. Some values are generated with Terraform using the `random` provider, some set to place by hand and some set by the `seed` Terraform. Program 3.1 presents how to generate a secret value using Terraform's `random` provider and set the generated value to secret manager secret version.

```
resource "random_password" "password" {
  length = 100
}

resource "google_secret_manager_secret" "secret" {
  project = "example_project"
  secret_id = "example_secret"
}

resource "google_secret_manager_secret_version" "version" {
  secret = google_secret_manager_secret.secret.secret_id
  secret_data = random_password.password.result
}
```

Program 3.1. *Creating a secret with value generated using Terraform.*

The main problem with using the method presented in Program 3.1 is that the password generated with `random` provider is stored in plain text in the Terraform state file [39]. This is by design, in order to allow Terraform to remember the value and not regenerate it on every apply [48]. This introduces an unnecessary security risk, where the password is available for anyone with access to the Terraform state file. Fortunately the Terraform state file is stored in a private, encrypted storage bucket and therefore is well protected. Nevertheless, there is no reason why the value should be stored anywhere else but the secret manager. To circumvent this minor vulnerability all secret values should be generated and set by hand, either by using GCP command line tool `gcloud` or with the web user interface.

The `seed` should manage secrets which are needed for the IaC pipeline to work. Currently, this is not the case as `seed` also manages a private NPM repository secret which is needed for building the application images. Therefore, managing of the secret should be moved to the environment Terraforms as it is not needed to run the IaC code.

Decoupling the handling of the secrets from the IaC allows for greater control over the secrets, as they are not managed by an IaC pipeline. When manually managing the secrets, new versions can be created at will without the need for IaC updates. The secret values can be generated with the tool of choice, as it is not tied to Terraform's `random` provider. It is also easier to manage secrets from external sources, as they can be added

directly to secret manager. There is no need for additional steps such as adding the secrets to a Terraform variable file, which increases the risk of accidentally adding the values to version control or leaving the secrets unprotected on the developer's file system.

Common ways to pass secret values to the application are the file system and environment variables, of which the environment variable approach is also in use in this project. Both of these are prone to some vulnerabilities. When secrets are available on the file system, possibly in commonly known files such as `.env`, some application vulnerabilities become more feasible, namely directory traversal attacks. Consuming secrets through environment variables is susceptible to misconfigurations and dependencies which may log environment variables, thus leaking them. Google suggests using a secret manager library to avoid these weak points, but using a cloud provider specific libraries ties the application implementation down to that provider. [55]

3.2.2 Using Cloud Build for CI/CD

The application images are currently built with Google Cloud Build triggers which run Google's Docker cloud-builder to build the image and then push it to Artifact Registry. The current solution is a working one, but not without its shortcomings. Cloud Build does not provide an easy way to tag application images with version numbers. Therefore, the current solution is to tag the images with the Git commit hash from which the image was built, so that the application can later be deployed using that hash. Using Cloud Build for CI/CD also creates a large amount of Cloud Build triggers, some of which are run automatically and some manually. Separate triggers are needed for each application on each environment and for both deployment strategies, manual and automatic.

It is quite a manual solution, as the versioning and deployment is controlled using Git branches, tags and pull requests, but the actual CI/CD pipeline is not on GitHub. Instead of simply clicking a *deploy* button on GitHub, the developers have to copy the hash of the correct commit, select the correct trigger from a list of 72 triggers on the Google Cloud Console before running said trigger with the hash. This creates a lot of manual and error-prone work. Because the same images are deployed onto each environment, the Artifact Registry and the Cloud Build triggers are located in the `infra` project. This means that triggers for all the environments are in the same listing, requiring the developers to be very careful of what they are doing.

One additional benefit of moving at least the build steps away from Cloud Build would be the absence of the need to pass the private NPM repository secret to Secret Manager, as described in Subsection 3.2.1, thus removing secrets completely from the `seed` Terraform. Writing GitHub actions instead of Cloud Build triggers with Terraform would also provide a better developer experience.

However, to be able to move the pipeline from Google Cloud Build to GitHub would require further development of the release process and the required GitHub Actions, which are not in the scope of this thesis.

3.2.3 Module usage

Each of the environments is constructed using shared Terraform modules. As already briefly covered in Section 3.1, there are multiple types of modules currently in use. Some modules are well implemented reusable building blocks, others have little reuse value. A good example of a well implemented module is a module called `cloud-run-api`. It is used to create all Cloud Run instances in the projects, their specific secrets and possible storage buckets.

Other modules on the other hand are merely a collection of loosely related, specific resources, which have little reuse value. An example of this is the multiple modules related to creating various Google Pub/Sub queues and their accompanying resources, such as publishers and dead-letter queues. Currently, there are three almost identical modules creating different queues with slightly different configurations, totaling over 200 lines of excess Terraform code.

Program 3.2 presents the main content of a module creating a single Pub/Sub queue. The module in the example has already been generalized and parameterized, meaning all the queues could be created using this single module. Program 3.2 is a good example of a well implemented module, which has no hard-coded specifics and can be used for all queue needs.

```
resource "google_project_service_identity" "sa" {
  provider = google-beta
  service  = "pubsub.googleapis.com"
}

resource "google_project_iam_member" "dlt_roles" {
  for_each = toset(["roles/pubsub.publisher", "roles/pubsub.subscriber"])
  role     = each.key
  member   = "serviceAccount:${google_project_service_identity.sa.email}"
}

resource "google_pubsub_topic" "topic" {
  name = var.topic
}

resource "google_pubsub_topic" "topic_dlt" {
  name = "${var.topic}-dead-letter"
}

resource "google_pubsub_subscription" "subscription" {
  name     = "subscription"
  topic    = google_pubsub_topic.topic.name
}
```

```

push_config {
  push_endpoint = var.push_endpoint

  oidc_token {
    service_account_email = var.endpoint_sa
  }
}

dead_letter_policy {
  dead_letter_topic = google_pubsub_topic.topic_dlt.id
}

resource "google_pubsub_topic_iam_member" "publisher" {
  count = length(var.publishers)
  topic = google_pubsub_topic.topic.name

  role   = "roles/pubsub.publisher"
  member = "serviceAccount:${var.publishers[count.index]}"
}

resource "google_pubsub_subscription_iam_member" "subscriber" {
  subscription = google_pubsub_subscription.subscription.name

  role   = "roles/pubsub.subscriber"
  member = "serviceAccount:${var.endpoint_sa}"
}

```

Program 3.2. Contents of a module creating a Pub/Sub queue.

Program 3.2 creates the necessary service account bindings for Google to use the dead letter topic, creates the topics themselves, a push subscription for the end point consuming messages from the queue and correct IAM privileges for the appropriate service accounts for publishing and subscribing to the queue. As seen by the example, the Terraform for creating the queue is easily parameterized, allowing for proper module usage instead of creating separate modules for each of the queues. It is notable that Program 3.2 example is quite simplified, using mostly Google's default configurations for the resources. In reality, the module would take in more variables than just the needed service account emails (`publishers`, `endpoint_sa`), end point addresses (`push_endpoint`) and topic names (`topic`) present in the example. The module could also take retry back-off configurations to fine tune how long a failed message is retried and message retention periods to determine for how long the messages are stored, to name a few.

Modules are also used for creating the Cloud Build triggers discussed in Subsection 3.2.2, as all the images need their specific triggers for all environments. Most of the triggers are created with a module called `cloud-build-trigger`, which creates triggers for automatically building and deploying to test environment, manually deploying to other environments and manually building the image from a specific branch. However, there are some images which need to be deployed to multiple Cloud Run instances using different configurations. For these applications there are image-specific modules, which are not

reusable in other contexts. The same pattern of specific modules also repeats in creating schedulers, which all have their own specific hard-coded modules instead of a single configurable module.

3.2.4 Declaring new resources and managing environments

In this project the environments are declared with their own Terraform configurations, composed of the modules discussed in Subsection 3.2.3. Because of this, all new resources need to be declared thrice, once to each environment. This causes unnecessary, error-prone manual work, especially if the new resources use a lot of configuration and references to other resources. The current approach to declaring environments is further visualized in a simplified form in Figure 3.2, where each arrow in the figure represents some form of dependency.

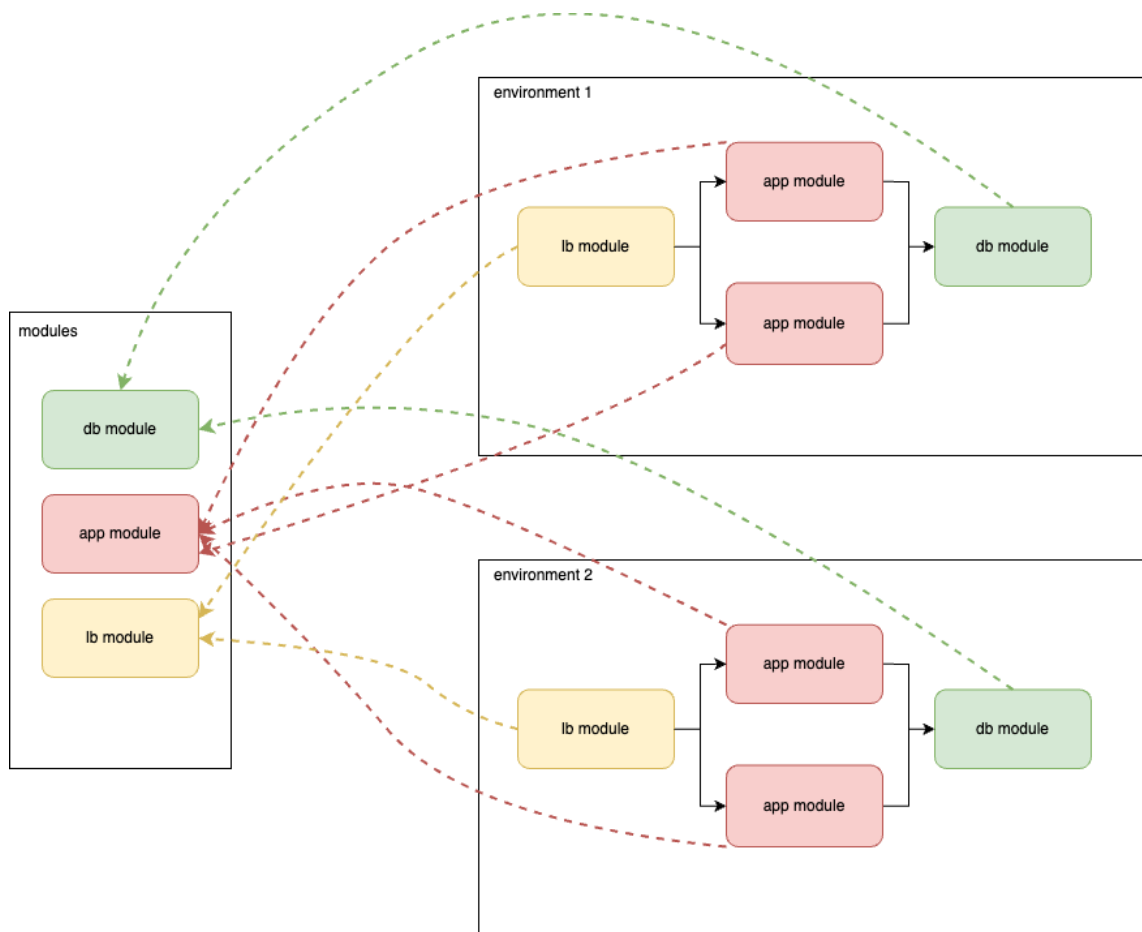


Figure 3.2. Current approach to using modules when declaring environments.

As can be seen from, Figure 3.2, the environments directly use singular modules. Because the environments are supposed to be near identical clones of each other, only differing in configuration, all changes need to be made separately to each environment. This means that when for example a new Cloud Run instance is introduced, all the envi-

ronment IaC files need changes. Furthermore, references inside the environments need to be made separately. For example, in Figure 3.2, both of the application instances declared with the app module use the same database declared with the db module, so the references to the database would need to be explicitly made to each environment separately. Adding a new resource, for example another application instance using both load balancer and database, would require first declaring the resource twice and then adding the appropriate references inside the environments twice. The application would also most probably need some configuration which is identical between the environments, suggesting it should be declared only once and not separately for each environment. Most of these configurations should be included in the module itself, but it might be feasible to have some configurations which stay the same between environments but differ on an instance basis. Example of this would be a processing heavy application, which would need the same amount processing power in each environment but a lot more than all the other applications declared with the same module, creating a need to declare configurations outside the module, but below the environment-level of IaC.

There are no guarantees that all these changes would be identical, as the manual work and thus possibility for errors multiplies with each environment. This is the biggest challenge what currently comes to the upkeep and maintenance of this infrastructure. Because of the current approach, each of the environment Terraform files are some 1300 lines long, which makes changing the existing infrastructure quite arduous. One possible solution would be to introduce another layer of Terraform, a stack of modules which would require only the actual environment specific configurations. The stack would then be used in each of the environments instead of separate modules. The stack-approach is further examined in Chapter 5 and Chapter 6.

3.2.5 Miscellaneous

The current infrastructure also contains other more miscellaneous development targets. The goal is to address all of these in the thesis in addition to the more severe issues introduced in previous sections.

Terraform and providers should be versioned in a `terraform`-block living in the environments root Terraform file, or in a separate `terraform.tf` file parallel to the environment root file. [60] Currently neither Terraform nor the providers are versioned, which may cause issues with updates later in the lifecycle of the infrastructure.

Another more seldom surfacing issue is passing a Cloud Run instance's URL to itself. One such situation would be when the application calls an API and needs to include its own URL in the payload in order for the API to asynchronously call a callback end point of the original caller. However, Terraform forbids resources from referencing themselves, so the URL can not simply be passed as an environment variable. Program 3.3 tries to pass

its own URL to the application running inside the Cloud Run instance using environment variables but results in an error during Terraform plan.

```
resource "google_cloud_run_service" "api" {
  template {
    spec {
      containers {
        image = "some-image"
        env {
          name= "SELF_URL"
          value = google_cloud_run_service.api.status[0].url
        }
      }
    }
  }
}
```

Error: Self-referential block

```
on ../../modules/cloud-run-api/main.tf line 114, in resource "google_cloud_run_service" "api":
114:         value = google_cloud_run_service.api.status[0].url
```

Configuration for google_cloud_run_service.api may not refer to itself.

Program 3.3. Cloud Run instance referencing itself.

The issue also persists on module level, where referencing a module output within said modules declaration results in an error presented in Program 3.4

```
Error: Cycle: module.api.var.env_vars (expand),
module.api.google_cloud_run_service.api, module.api.output.url (expand)
```

Program 3.4. Error message produced by a module referencing itself.

One way to subvert this issue would be to first create the resource with a placeholder value for the URL, and later setting the correct value in place with a standalone change that is executed separately. The URL of a Cloud Run instance is visible in GCP web console after the resource has been creating in the first step. However, because of the Git flow in use, all the infrastructure changes are merged together when changes are made to the test environment. This causes the issue to still persist in the staging and production environments, even though the changes were separately made to test.

Some storage buckets are declared directly on the environment level, some inside the cloud-run-api. Buckets declared on the environment level should be moved to the cloud-run-api declaration of the Cloud Run instance using said bucket, in order to unify the way storage buckets are declared in this project.

Lastly, in order for the Cloud Run instances to call other instances via HTTP requests, they need a IAM role roles/run.invoker, which gives them the right to invoke other instances. [9] However, the current way how roles/run.invoker is set does not specify which other services the role-holder can invoke. There is a need for more specific access control, so that services can only call specific services that they have access to.

4. MOTIVATION AND RESEARCH QUESTIONS

The main motivation for this thesis is to improve the maintainability, developer experience and further development possibilities of an existing, production-use infrastructure. As discussed in previous sections, the infrastructure contains multiple shortcomings, which should be addressed in order to improve maintainability and reduce the need for tedious manual work. The goal is to address all the issues introduced in Chapter 3. In addition, this thesis tries to compile best practices related to working with IaC, Terraform and Google Cloud through the process of improving the current infrastructure.

Software maintainability is defined in [34] as the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. Popular metrics for demonstrating software maintainability are technical debt and code smells. Technical debt explains the consequences of non-optimal software development decisions on a project. By taking technical debt, developers can take shortcuts to save time, but the accumulated debt must be paid and often gains interest over time. [8] Code smells are surface indications that usually correspond to more significant problem in the software [26]. This thesis focuses on paying back technical debt and addressing code smells to improve the maintainability of the case project infrastructure.

Developer experience is defined as how easy or difficult it is for a developer to perform the essential tasks in order to make a change [18] or as the interactions and feelings the developer has when working with code to meet an objective [49]. This thesis tries to improve the developer experience of the case project infrastructure by reducing manual work and increasing the quality of code through refactoring.

The main motivation for the thesis is molded into two research questions:

RQ1 How to improve the maintainability and developer experience of IaC?

RQ2 How to make fundamental changes to IaC after going to production?

The research questions are used to guide the implementation when addressing the issues with current infrastructure. RQ1 is the main question to which this thesis tries to answer, and in order to achieve that, RQ2 must also be answered.

Answers to the questions are sought through a constructive research approach. Kari

Lukka [38, 40], one of the authors behind the approach, describes constructive research approach as a methodology producing innovative constructions aiming to solve real life problems and thus making contributions to that field of science where it is applied. According to Lukka main steps of the process are gathering a deep understanding of the subject, developing, implementing and testing a solution and finally analyzing both the applications and the theoretical contributions of the solution. All the steps of the approach are introduced in Figure 4.1.

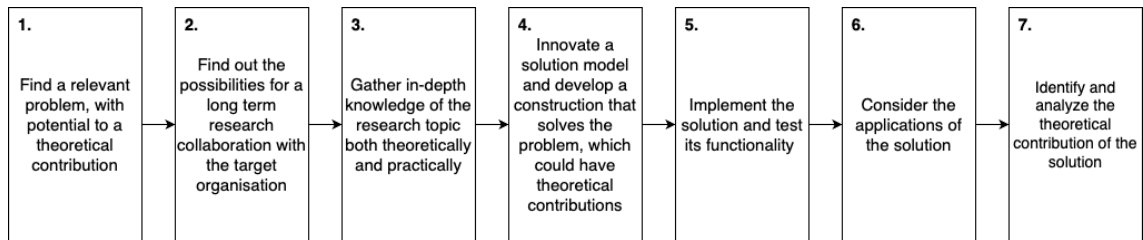


Figure 4.1. *The process of constructive research approach, modified from [37], originally based on [40].*

In practice this means researching the topic, planning and then carrying out concrete changes and improvements to the infrastructure, and observing their effects. Made changes and their effects are evaluated by the development team of the case project, with the goal of finding out if the changes made in this thesis improved the infrastructure. Best practices and answers to the research questions found during the research and implementation of the solution represent the theoretical contribution of this thesis.

5. IMPLEMENTATION PLAN

This chapter presents the planned changes to the existing infrastructure to be made within this thesis. Changes are carried out as part of normal development process in the project, as individual pull requests. It is worthy to note that deployment of some changes may require a service break as the changes can not be rolled out seamlessly without disruption to the service. Implementation plan is divided into discrete action points described below in no particular order.

AP1 Refactor modules to be as universal as possible

AP2 Introduce a *stack* layer to the infrastructure: a Terraform module that is composed of the whole current infrastructure

AP3 Remove all instances of secret value generation via Terraform, create secrets with Terraform and values by hand

AP4 Develop a deployment trigger deploying all images to staging or production accordingly, to ease the release process

AP5 Version Terraform and providers properly

AP6 Tighten IAM roles around invoking Cloud Runs

AP7 Research and implement a way for Cloud Run instances to reference themselves if possible

Action point 1 includes modifying the IaC code declaring the resources. Modifying existing resources may cause Terraform to destroy and recreate the resource when executing the changes, causing disruption to the services. In case of schedulers, this may lead to skipping a scheduled job. To ensure that no scheduled job is skipped, the changes should be deployed outside the known times when schedulers are run. In case of Pub/Sub queues, messages may be lost if they are yet to be processed. Therefore, old queues should exist parallel to the new ones with a transition period, in order to make sure all the messages in the old queues are processed before they are destroyed. With most other resources the possible recreation of the resources is not a problem, if the deployment is scheduled to a low traffic point in time or a service break is used.

Action point 2 poses more challenges, as the fundamental way of how resources are declared is changed. Terraform may recreate all resources despite the end result being

identical to the starting point. Most critical resources are database and storage buckets, as they contain data that should not be lost. In case it is noticed that deploying the changes would lead to recreating of some resources containing data that would be lost, old and new versions of the resource need to exist parallel and the data transferred.

In case of storage buckets, the data can be simply transferred from one bucket to another using tools `gsutil` or `gcloud`, if the data volume is smaller than one terabyte. [45, 57, 64] For the database, the data may be transferred from one Cloud SQL instance to another by replicating the database [13] or using database dumping methods [23, 24]. Transferring the database data is not in the scope of this thesis and should be avoided if possible.

Because the stack is introduced to an already existing infrastructure, the easiest way to ensure data preservation is to leave these critical resources outside the stack. This is a feasible approach when it comes to database, as its declared using a standalone module. However, storage buckets are created by the `cloud-run-api` module which also creates Cloud Run instance. This practically means that buckets can not be easily left outside the stack, if the usage of said modules continues. Therefore, a data transfer is needed for storage buckets.

Action point 3 should also be implemented with the old implementation existing parallel to the new one, in order to provide a possible fallback option if something does not work out when switching implementations. Action points 4, 5 and 6 are considerably more simple than the former and should not require any special actions, action point 7 requires through research in order to find out if such change is even possible to make.

6. IMPLEMENTATION

This chapter further examines the action points introduced in Chapter 5 and presents the process of acting on the action points. Furthermore, this chapter showcases the state of the infrastructure after the changes made in this thesis. Note that the presented code examples might not be complete and working as is, as they may have been truncated for brevity.

6.1 Action point 1: Refactor modules to be more universal

The case project currently contains multiple poorly built Terraform modules, discussed in Subsection 3.2.3. A good example of a well-implemented Terraform module is the example module creating Pub/Sub queues, introduced in Program 3.2. The module takes in changing configuration as variables, handles creating the needed resources such as service accounts and IAM role bindings, and utilizes loops for creating the resources dynamically with minimal Terraform code and no hard coded values. The example module acts as a baseline which the other modules should fulfil. Already well-implemented modules in the case project are `cloud-task-queue`, which handles creating Cloud Task queues and `cloud-run-api`, which creates Cloud Run instances, their secrets and storage buckets. Modules in need of a refactoring are modules creating queues, schedulers and modules creating Cloud Build triggers, as they contain a lot of repeated and near identical Terraform code.

6.1.1 Pub/Sub modules

The refactored Pub/Sub module follows the example discussed in Subsection 3.2.3 and presented in Program 3.2. In addition to the example, the module takes in more variables, such as back-off and retry configurations determining how many times and how often a message is retried if its delivery fails, and retention configurations determining for how long the messages are persisted in the queue. By default, the values in this project are 10 delivery attempts with minimum backoff of 30 seconds and maximum backoff of 10 minutes, retention in the topic for one day and in the dead letter topic for five days.

An issue was discovered in the module, when executing the infrastructure changes creat-

ing new Pub/Sub queues. The module expects publishers to be a set of strings, i.e. a list of unique strings. However, if these values are dynamical, for example references to service accounts, the following error occurs when trying to apply the changes.

```
Error: Invalid for_each argument
```

```
on ../../modules/pubsub-queue/main.tf line 84, in resource
    "google_pubsub_topic_iam_member" "processing_topic_publisher_member":
84: for_each = var.publishers

var.publishers is set of string with 3 elements
```

The "for_each" set includes values derived from resource attributes that cannot be determined until apply, and so Terraform cannot determine the full set of keys that will identify the instances of this resource.

When working with unknown values in for_each, it's better to use a map value where the keys are defined statically in your configuration and where only the values contain apply-time results.

Alternatively, you could use the `-target` planning option to first apply only the resources that the for_each value depends on, and then apply a second time to fully converge.

Program 6.1. *Error message when applying changes containing a new Pub/Sub queue.*

The error presented in Program 6.1 occurs when trying to apply changes creating a Pub/Sub queue and its publisher Cloud Run instance at the same time, with the references already in place. Changing the module variable `publishers` from a set of strings to a map of strings fixes the issue, as now the publishers have hard-coded keys and Terraform does not have to rely on apply-time results.

6.1.2 Cloud Scheduler modules

The case project currently contains three modules creating Google Cloud Scheduler jobs. Each of these modules targets a single Cloud Run instance and contain multiple scheduler jobs. The scheduler service accounts are given `roles/run.invoker` using Terraform resource `google_project_iam_member`, which allows them to call not only the intended service, but all other services present in that project, too. As a part of the change discussed in Section 6.6, the scheduler module no longer sets IAM privileges, but outputs its own service account which is then given to the correct Cloud Run module for permissions. The change is introduced in more detail in Section 6.6.

The main challenge in refactoring multiple Cloud Scheduler modules into one universal module is the need to create multiple configurable scheduler jobs. This is achieved by typing a complex variable, which defines multiple jobs. The jobs are then created utilizing a loop looping over the variable containing the definitions. Creating said variable is presented in Program 6.2.

```
variable "jobs" {
  type = map(object({
```

```

    description = string
    schedule    = string
    endpoint    = string
    body        = string
  )))
  description = "Map of objects defining the jobs
to be created for this scheduler. Note that the map key must be shorter than
29 characters because it is used as a part of the job resource name"
}

```

Program 6.2. *Typed variable defining jobs to be created for the scheduler.*

The variable presented in Program 6.2 is then used in Terraform meta-argument `for_each` in the resource block creating the jobs. When a `for_each` meta-argument is set for a block, an object called `each` is available with two attributes, `key` and `value`. [62] The `key` is used as the job name in this context, with the `value` containing the job definitions according to Program 6.2. Using the `for_each` meta-argument is presented in a truncated example in Program 6.3.

```

resource "google_cloud_scheduler_job" "scheduler_job" {
  for_each = var.jobs
  name     = each.key
  schedule = each.value.schedule

  http_target {
    http_method = "POST"
    uri         = each.value.endpoint
    body        = base64encode(each.value.body)
  }
}

```

Program 6.3. *Creating scheduler jobs using a `for_each` meta-argument.*

In addition to the example presented in Program 6.3, each job takes in common variables, which are defined at the scheduler level, meaning they are the same for each job in the scheduler. The variables are similar to the configurations defined for Pub/Sub queues in Subsection 6.1.1, such as `backoff` and `retry` configurations. An example of using the new scheduler module is presented in Program 6.4.

```

module "scheduler" {
  source = "../../modules/cloud-scheduler"

  project = data.google_client_config.current.project
  location = data.google_client_config.current.region

  sa_short_name = "scheduler"
  sa_display_name = "This is an example scheduler."

  jobs = tomap({
    job1 = {
      description = "Job 1 doing things",
      schedule    = "30 9 * * *",
      endpoint    = "https://www.example.org/job1-endpoint",
      body        = "{}"
    }
  })
  job2 = {

```

```

description = "Job 2 doing stuff",
schedule    = "45 9 * * *",
endpoint    = "https://www.example.org/job2-endpoint",
body        = "{ \"key\": \"value\" }"
}
}))
}

```

Program 6.4. *Creating a scheduler with multiple jobs using the module.*

Using the module as in Program 6.4 would create two scheduler jobs, named `job1` and `job2`. The module takes in the schedule for the job in cron job format [14]. In the example, the jobs are scheduled to run at 9:30 and 9:45 server time. In addition to the schedule, each job must have an end point which the scheduler calls, and a possible body. This module makes only POST requests, and as such the body needs to be always defined, although it may be empty, like in Program 6.4 `job1`. It would be possible to create all scheduler jobs using a single instance of this module, if the privileges of the scheduler were no concern and the scheduler had the right to call any service. However, as part of the changes of Section 6.6, one scheduler should have the privileges to call only the services which it concerns, and no other services. For this reason, the schedulers should be separated per service, instead of creating one all powerful scheduler.

6.1.3 Cloud Build trigger modules

Currently, there are three different modules creating Cloud Build triggers. One of the modules is a universal module, which is used for most Cloud Build trigger needs. However, two of the modules are image specific, as the images are deployed to multiple Cloud Runs. The universal image can currently only deploy to one Cloud Run.

The idea was to refactor the universal Cloud Run module to utilize a variable called `deploy_targets`, which would define multiple Cloud Runs to which to deploy the image. Unfortunately, `build` blocks used in declaring build templates for the Cloud Build trigger does not support dynamic blocks [21]. Dynamic step block would allow declaring the steps based on the incoming variable, and thus eliminating the need for multiple modules where the steps are hard-coded. But as dynamic blocks are not supported, it leaves few options. A loop could be utilized on a higher level, creating entire triggers in a loop. This would come at a cost, as it would further multiply the amount of triggers in the infra project, as multiple deployment steps could not be included in a single trigger anymore. Another solution would be to implement a custom Terraform generator, which would generate the wanted static Terraform code on demand. This would require substantial research and implementing, and as such is not in the scope of this thesis. Therefore, the Cloud Build trigger modules are best left as is, with the current universal module used for most needs and custom modules implemented for more specific needs.

6.2 Action point 2: Stack of modules

As previously discussed in Subsection 3.2.4 and illustrated by Figure 3.2, declaring new and modifying existing resources currently requires too much manual and therefore error-prone work. Solution to this problem is a Terraform architecture pattern *stacks*, presented in [43]. *Stacks*, or *stack-of-modules*, architecture introduces an intermediate layer to Terraform code. Instead of each of the environments directly declaring standalone modules and resources, potentially leading to thousands of lines long environment files, modules and individual resources are declared in a *stack*, which is then reused in each of the environments. There is no limit for the number of stacks, but in this project only one stack is implemented, containing most of the infrastructure. Consider the stack-of-modules architecture illustrated in Figure 6.1 compared to the previous implementation presented in Figure 3.2.

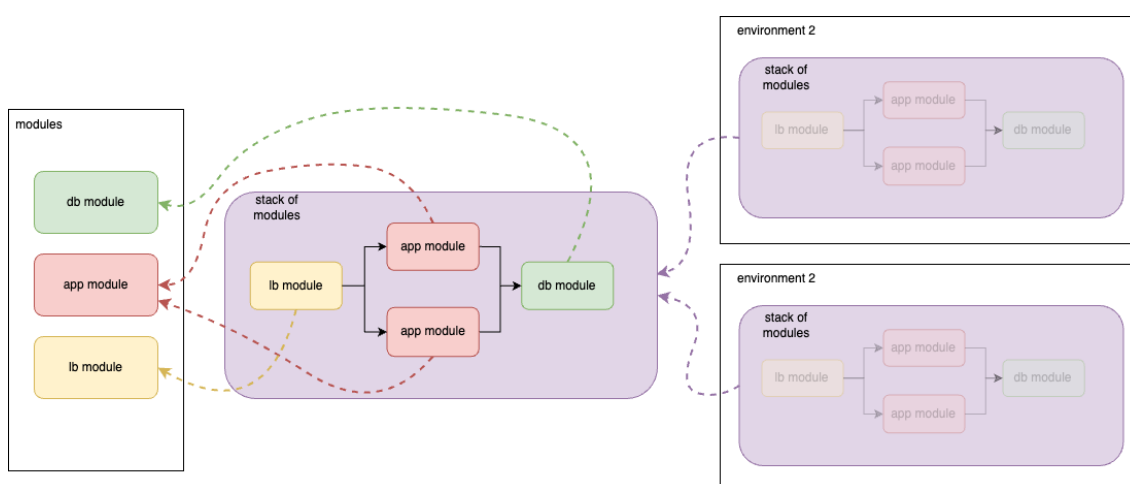


Figure 6.1. Stack-of-modules approach when declaring environments.

As can be seen from Figure 6.1, the number of arrows has greatly reduced, indicating that there is less manual references and declarations to be made. In the optimal case when declaring new resources, only the stack has to be modified. In case the new resource requires environment specific configuration values, the correct values should be set in each of the environments. The stack-of-modules architecture leads to much leaner environment files, as they contain only the stack declaration, with only the environment specific configuration. The stack will still potentially be thousands of lines long, but now there is only one such file instead of multiple near identical, massive files.

Implementing the stack begins by picking the relevant resources from one of the existing environment files into their own module. Resources that are going stay outside the stack should not be copied. Given the size of the environments, the easiest way is to copy the entire content of one of the environment files and removing parts that should stay outside the stack. In this instance, the database and all direct dependencies of it, such as virtual

private network configurations, are going to stay outside the stack.

The stack module is going to live parallel to `environments` and `modules` folders in the directory structure introduced in Section 3.1, in a folder called `stack`. The stack module includes files `main.tf`, `variables.tf` and `outputs.tf`. `main.tf` contains the stack implementation, i.e. resources which previously were declared directly in the environment files. `variables.tf` declares the environment specific variables, which should be set in each environment when declaring the stack. `outputs.tf` declares the outputs of the stack, which there currently are none. From the copied environment declaration, variables changing by the environment should be added to `variables.tf` and referenced accordingly.

```
// Declaring variables in module/variables.tf
variable "project" {
  type      = string
  description = "Current GCP project"
}

// Referencing a variable in module/main.tf when declaring a resource
resource "google_cloud_run_service" "api" {
  project = var.project
}

// Passing a value to a variable when using the module
module "module" {
  project = "gcp-project-name"
}
```

Program 6.5. Passing, declaring and referencing module variables

Once all the required variables are declared and referenced as shown in the example Program 6.5, correct values should be gathered from the old environment declaration and set to place. However, figuring out which variables should be declared in the stack's `variables.tf` is a lot of manual work. A quick way to find out which values are changing between environments is to run a differential between the environment files. This can be done for example using Git, as shown in Program 6.6.

```
git diff --no-index [--] <path> <path>
```

Program 6.6. Git diff comparing two files outside a Git working directory [28].

The variable values can also be gathered from the output of the Git command introduced in Program 6.6. Example output of Git diff is displayed below.

```
- branch_regex = "^develop$"
- branch       = "develop"
- env          = "test"
+ branch_regex = "^release$"
+ branch       = "release"
+ env          = "stg"
```

Program 6.7. Example output of Git diff between test and staging environment files.

As can be seen from the example Program 6.7, variables `branch_regex`, `branch` and `env` have differing values between environments, suggesting that they should be declared in the stack's `variables.tf` and the values should be set from outside the stack. Fortunately, in this project's environments most of the environment specific configurations are declared as Terraform local variables. Instead of adding each of the variables separately to stack's `variables.tf`, the whole environment specific locals object can be typed and added as one variable, saving much of the manual work. In addition, referencing this configuration object is similar to referencing the locals, as the object structure stays the same. For example, accessing the database name in locals with `local.database.app1.name` changes to `var.environment_configuration.database.app1.name`, meaning only minor changes are needed inside the stack.

Once the references are updated inside the stack, and all the necessary variables are added, the stack can then be declared in the environment file. Critical resources, such as queues and buckets, of the old implementation should exist parallel with the stack for a while. Queues need a transition period in order to make sure no messages are lost. In practice this means that old and new queues exist concurrently, with new messages being pushed into the new queue, but messages are being processed from both queues. This way all the messages in the old queue get processed before the resource is destroyed. This means that the new application instances need to process messages from the old queues also, until the end of the transition period, after which the old queues are destroyed. Because the new application instances now live inside the stack, the end points and service accounts of the processing applications need to be exported from the stack by declaring them in the stack's `outputs.tf` file. This way the old queue living outside the stack can push messages to the new handlers during the transition period, making sure all messages get processed. Regarding schedulers, they can be paused for the duration of infrastructure changes to make sure no runs are attempted while the changes are in progress.

When the stack exists alongside old infrastructure, data needs to be transferred from the old buckets to the new. This is possible via a simple command in Google's `gcloud` tool, when transferring less than one terabyte [64].

```
gcloud storage cp --recursive gs://SOURCE_BUCKET/* gs://DESTINATION_BUCKET
```

Program 6.8. *Example command copying contents from one Google Storage Bucket to another [45].*

The command presented in Program 6.8 takes a few seconds to transfer the data. The command needs to be run for all the buckets separately. To reduce the risk of accidentally losing the data, an intermediate bucket can be created manually without Terraform and the data transferred there before creating the final buckets using the stack. This way if something goes wrong, the data stays untouched as Terraform has no control over

resources which it did not create. This also enables removing the old infrastructure with the same infrastructure change as creating the new one, as the data is temporarily moved to the intermediate bucket.

The same issue occurs with secret values. Because `cloud-run-api` module controls secrets related to the application running in the Cloud Run created by the module, when introducing the stack all secrets are recreated and their values lost. To mitigate this, values should be either moved to intermediate secrets for the transition or made sure that the values are available elsewhere and not lost.

GCP resources have to have unique names. If both old and new infrastructure are to exist parallel, resource naming needs to be updated. Using intermediate buckets and secrets allows for destroying the old resources when creating the new ones, so there is no conflict. But in case of the queues, in order to make sure that no messages are lost, both queues need to co-exist. Easiest way to achieve this is to update the naming scheme in the module creating the queues, making sure that all the new queues have different names than the old ones.

6.3 Action point 3: Remove all instances of Terraform secret value generation

As discussed in Subsection 3.2.1, the case project currently contains some secret values which are generated with Terraform. This is not optimal, and the development team has decided to forgo this approach and instead manage all secret values by hand.

When removing the Terraform declaration for a secret value, the version is destroyed. Therefore, a new secret version should be manually created using Google Cloud console before removing the secret version resources from Terraform code. This way there are no disruptions to the services consuming the values, as a secret version is always available. Once the values are available in new versions, the old Terraform generated versions are simply removed from the Terraform code, destroying the version along its generated value. Only the secret version should be deleted, there is no reason to delete the secret itself. By only creating a new version for the existing secret all the references throughout the Terraform code stay intact and require no changes.

After deleting the generated values, it would be good practice to rotate the values, so that the old values are no longer used. For most secrets this is only a matter of generating or fetching a new value and creating a new secret version with it, but the database requires further actions. This is because the password must be set to the PostgreSQL database user too, in addition to updating the value in secret manager. Possibilities for updating the password for the PostgreSQL user are either creating a migration which alters the user, or connecting manually to the database and altering the user by hand. In this project

the database users are handled manually, as standard PostgreSQL users without any integration to Google Cloud IAM for example.

```
ALTER ROLE <username> WITH PASSWORD '<new password>';
```

Program 6.9. *Update a database user password in PostgreSQL [2].*

Program 6.9 presents an example command altering a database user's password in PostgreSQL. The command can be run as is when connecting to the database from developers own terminal, given the developer has sufficient privileges and the connection is possible. However, this is not advisable, as the value would be then sent to the server unencrypted. A `psql` command `\password [username]` should be used instead, as it prompts the developer for new password and encrypts the value before sending it to the server as an `ALTER` command. [2, 47]

Another option would be to create a migration containing the same command, which is then executed by the appropriate Cloud Build trigger. The migration trigger already contains bindings to the correct secrets, so only a new version is required for the secret containing the password. The migration should be ready to be executed as soon as the new version is added, because all bindings to the secrets use the latest value. This means that the services using the latest value will not be able to log in to the database before the migration is executed and the user altered. An example migration file using the `node.js` library `node-pg-migrate` is presented in Program 6.10.

```
exports.shorthands = undefined

exports.up = (pgm) => {
  pgm.sql("ALTER ROLE {username} WITH PASSWORD '{password}'", {
    username: example-db-user,
    password: process.env.DB_PASSWORD
  })
}

exports.down = (pgm) => {
  pgm.sql("ALTER ROLE {username} WITH PASSWORD NULL, {
    username: example-db-user
  })
}
```

Program 6.10. *A node-pg-migrate migration file altering a role, based on [2, 54].*

The *down* migration in the migration file in Program 6.10 does not revert the password back to the old password, but rather resets it altogether. Therefore, it is all the same to run the *up* migration again with an updated value if the first run did not yield the wanted result, and the value needs to be updated.

In addition to updating the password value to the database and secret manager, the containers running in the Cloud Run instances must be rotated. In the project secret values are bound to Cloud Run's environment variables using the latest version. Because the secrets are passed to Cloud Run instances using environment variables, they are

resolved at instance startup time [66]. This causes the variables to update when the container stops and starts, which may cause issues if the old value is no longer used in the database. Therefore, when rotating the secret values, all containers should be redeployed to make sure they use the latest version of the secrets. The universal deployment trigger developed in Section 6.4 makes this a one-click operation.

6.4 Action point 4: Develop a universal deployment trigger

As discovered in Subsection 6.1.3, Terraform does not support dynamically declaring build steps for Cloud Build triggers. This rules out most elegant solutions which would reduce the amount of code. But as this universal deployment trigger is still a very much needed and wanted feature, it is implemented mostly statically, by hand. This is by no means according to any good practices, but the impact this trigger has on the deployment process is worth some bad code.

To aid the process of declaring all deployment steps by hand, the images and associated Cloud Run names are first gathered as `locals`. In addition to defining the static image and Cloud Run names to `locals`, the common parts of a single deployment step can also be exported to `locals`. This way the actual steps are considerably shorter, and it is easier to read what it actually does. The `locals` for the universal deployment trigger deploying one image to two Cloud Runs, and the separated static parts are presented in Program 6.11.

```
locals {
  // Locals defining image and Cloud Run names
  app1 = {
    image = "app1"
    cloud_runs = {
      run1 = "app1-run1"
      run2 = "app1-run2"
    }
  }
}

// Reused bits of a deploy step to make the triggers leaner
command = ["run", "services", "update"]
flags = [
  "--platform=managed",
  "--project=${_TARGET_PROJECT}",
  "--region=europe-north1",
  "--labels=managed-by=gcp-cloud-build-deploy-cloud-run,
  commit-sha=${COMMIT_SHA},gcb-build-id=${BUILD_ID},gcb-trigger-id=${TRIGGER_ID}",
  "--quiet",
]
registry = "europe-north1-docker.pkg.dev/
  ${data.google_client_config.current.project}/docker-registry"
}
```

Program 6.11. *Locals for the Google Cloud Build trigger presented in Program 6.12.*

A Cloud Build trigger is then constructed using the `locals` from Program 6.11, with concatenating the hard-coded parts from the `locals` with the dynamic parts containing the

image and Cloud Run names. This is then repeated for all the Cloud Runs, until there is a step handling the deployment of each image into each Cloud Run included in the trigger. An example trigger showing how to construct the steps using said locals is presented in Program 6.12.

```
resource "google_cloudbuild_trigger" "example" {
  for_each = {
    stg = "release"
    prod = "main"
  }

  source_to_build {
    uri      = https://github.com/some-repository
    ref      = "refs/heads/${each.value}"
    repo_type = "GITHUB"
  }

  build {
    step {
      id = "deploy ${local.app1.image} to cloud run ${local.app1.cloud_runs.run1} on ${each.key}"
      name = "gcr.io/google.com/cloudsdktool/cloud-sdk:slim"
      args = concat(
        local.command,
        ["${_TARGET_PROJECT}-${local.app1.cloud_runs.run1}-run"],
        ["--image=${local.registry}/${local.app1.image}:${COMMIT_SHA}"],
        local.flags
      )
      entrypoint = "gcloud"
    }

    substitutions = {
      _TARGET_PROJECT = "project-name-${each.key}"
    }
  }
}
```

Program 6.12. Example Google Cloud Build trigger deploying an image to a Cloud Run.

The changing values in Program 6.12 are set to place using string templates. The values are given to the template as either direct Terraform reference or a Cloud Build substitution. Terraform references are used for values coming from Terraform data blocks, such as the project name when defining the registry in Program 6.12.

Terraform's string templates work similarly to most other programming languages, with `${ ... }` being an interpolation. Interpolation evaluates the expression between the markers and if needed, converts it to string before adding it to the surrounding string. [58] In Cloud Build triggers there are also predefined substitutions, such as `$COMMIT_SHA` which contains the hash of the Git commit on which the trigger is run. In addition to predefined substitutions, one can define their own by using a `substitutions` variable in the trigger. An example substitution is presented in the end of Program 6.12. To use self-defined substitutions, they must begin with an underscore and be referenced using a literal `${ ... }`, such as `$_TARGET_PROJECT`. [59] But, because the sequence

`#{ ... }` is already reserved for interpolation in Terraform, the characters must be escaped, resulting in `$${_TARGET_PROJECT}` as seen in Program 6.12. [58]

Similar steps than the one presented in Program 6.12 are added to the trigger for all Cloud Runs. Because the steps are defined to one trigger, the deployment of each Cloud Run is done in series [30]. This is not necessary as the steps do not depend on each other. Adding `wait_for = ["-"]` to each of the steps makes them run immediately at build time, instead of the default execution order [15]. Once all the steps are declared, the trigger can be used to deploy correct images with a given hash to all Cloud Runs with a single click. It is worthy to note that this solution is not very elegant and requires manually adding more steps when introducing new Cloud Run instances. However, despite being a subpar solution, it aids the deployment process greatly.

6.5 Action point 5: Version Terraform and providers properly

As briefly mentioned in Subsection 3.2.5, currently neither Terraform nor its providers are properly versioned. Terraform uses a format `major.minor.patch` for versioning, where minor and patch versions are backward compatible [41].

At the time of writing this thesis, the latest Terraform version is 1.5.0. Terraform is provided to the Cloud Build trigger as an image, and each of the steps can use a different image. For convenience, all the steps in the trigger use the same image. In this project the image version is defined in `terraform-cloudbuild.yaml`. The image version in use is currently 1.3.9, which is not too old. Updating the Terraform version which executes the IaC code is just a matter of increasing the version number. According to [65] updating from minor version 1.3 to 1.5 requires no actions, at least when considering the case project. Updating the Terraform requires no other actions than bumping the version number, but it is good to make sure that developers are using the same version locally when developing changes.

Now that the Terraform in use is the latest version, we can easily introduce version constraints for Terraform in the code, making sure the Terraform trying to execute the code is of correct version. It is suggested in [41] that the constraint is set using `~>` style version constraints, which pins the major and the minor version, but allows patch versions without modifying the constraint. This way the updates to Terraform are controlled, and it can be verified that the project still works on newer versions when updating. The version constraint is set to the `terraform` block of each environment, as presented in Program 6.13.

```
terraform {
  required_version = "~> 1.5.0"
}
```

Program 6.13. Terraform version constraint.

After the `required_version` is set, Terraform does not initialize the project if the Terraform version is not supported by the configuration. [41]

Providers on the other hand do not have currently any version number specified. By examining the Terraform lock file, it is seen that the current version for `google` providers is 4.55.0 in all environments. Both `google` and `google-beta` providers are version 4.32.0 in the *infra* project. Required providers can be set to the `terraform` block of each environment, much like the required Terraform version set earlier in this section. The latest version of both `google` and `google-beta` is 4.66.0 at the time of writing this thesis.

```
terraform {
  required_providers {
    google = {
      version = "~> 4.66.0"
    }
    google-beta = {
      version = "~> 4.66.0"
    }
  }
}
```

Program 6.14. Providers version constraint.

Program 6.14 presents how to declare the required providers and their versions in the `terraform` block. If the correct providers are not present when running `terraform init`, the initialization fails. The providers can be updated using `terraform init -upgrade`, which updates all providers and modules to the latest version which matches the constraint, which in this case is the latest release, version 4.66.0.

The upgrading of providers must be done by hand, as the `terraform init` command in `terraform-cloudbuild.yaml` does not include the `-upgrade` flag. This way the upgrading of providers is a manual and conscious effort, preventing accidentally or automatically updating something and possibly breaking things in the process. The Cloud Build trigger utilizes the Terraform lock file stored in version control to determine which versions should be initialized. When running `terraform init -upgrade` locally, the lock file is automatically updated.

In addition to versioning Terraform itself and its providers, modules too can be versioned. Currently, third party modules have a version constraint defined, and there is no need to change those. The project's custom, self-implemented modules do not have versioning, so the present version of those modules is always used.

6.6 Action point 6: Tighten the IAM roles around invoking Cloud Runs

Currently, the service account of a Cloud Run instance is given an IAM role `run.invoker`, which allows it to invoke other Cloud Run instances. The role however does not specify

which other Cloud Run instances the service account can invoke, but gives access to all other services. The role is given inside the `cloud-run-api` module, which takes an array of roles to give to the service account associated with the Cloud Run instance. From a security perspective, this approach results in excess privileges which should be avoided. The principle of least privilege is a widely adopted security practice, where the entity at hand is granted the least possible privileges necessary to perform its functions [56].

Instead of granting a project-wide role `run.invoker` via the Terraform resource called `google_project_iam_member` to each of the Cloud Run service accounts which need to call some other service, the service account of the invoker should be given an explicit privilege to call only the specific service it needs. This is achieved through the Terraform resource `google_cloud_run_service_iam_member`, which is a service specific version of the previously mentioned resource [33].

```
resource "google_cloud_run_service_iam_member" "invokers" {
  for_each = var.invokers

  project = var.project
  location = google_cloud_run_service.api.location
  service = google_cloud_run_service.api.name
  role    = "roles/run.invoker"
  member = "serviceAccount:${each.value}"
}
```

Program 6.15. Granting service-specific invoker roles.

Program 6.15 presents the granting of service-specific `run.invoker` roles inside the `cloud-run-api` module. The module takes a new variable `invokers`, which is a set of service account email addresses which should be granted the right to invoke this service. Note that because the roles are set using `for_each`, the resource used should be specifically `iam_member` as it is non-authoritative and does not overwrite the existing members of the role [33].

After adding appropriate service account emails to the Cloud Run declarations, existing `run.invoker` role definitions can be removed. The change is not very big from an implementation perspective, but all the more important from security and good practices perspectives.

6.7 Action point 7: Cloud Run instances referencing themselves

Another issue discussed in Subsection 3.2.5 is the inability for Cloud Run instances to reference themselves. As presented in Subsection 3.2.5, there is occasionally a need for Cloud Run instances to know their own URL, e.g. for callback uses. However, as discussed earlier, there is no straight forward way to achieve this.

The public URL of a Cloud Run instance follows a preset pattern, presented in Program 6.16

`https://<cloud run name>-<project hash>-<region>.a.run.app`

Program 6.16. *URL pattern for Cloud Run instances [35].*

The Cloud Run name and the region where the Cloud Run is deployed are known and thus could be hard-coded for new instances in need of a self-reference. The issue is the project hash, which is not predictable and is not available before a Cloud Run instance is created and its URL inspected. However, the URL is project specific, meaning stays the same inside one GCP project or in this case one environment. Therefore, if the hash is known from a previous Cloud Run instance, the URL can be deduced and hard-coded to circumvent issues related to missing environment variables or circular references.

This does not fix the issue, but provides a way to work around it, as there is no official way to fetch the project hash without creating a Cloud Run instance. When creating a Cloud Run instance in a project where the hash is not known, consider creating a dummy Cloud Run and fetching the hash that way, so that there is no need to modify the IaC code multiple times.

More proper way to handle the issue would be to add the Cloud Run instances behind a load balancer or an API gateway and letting it handle the routing. This approach however would create unnecessary overhead, if its only purpose was to handle minimal traffic between internal services. An internal load balancer or API gateway solution is a tradeoff between doing things by the book but managing more infrastructure or cutting corners for a leaner solution.

7. RESULTS AND EVALUATION

This chapter evaluates the changes made in Chapter 6 against the plan and the problems introduced in Chapter 3. The goal is to determine if the made changes solve any of the presented problems and actually help with the further development and maintenance of the infrastructure at hand.

This chapter also presents a set of best practices, compiled based on the results of this thesis and personal experiences working the case project. The list only concerns developer experience and maintainability side of Terraform IaC code, and is not a comprehensive guide on how to provision infrastructure using Terraform on Google Cloud.

7.1 Evaluation of the implementation

This section discusses the overall results of the changes made in this thesis. Overall the improvements implemented to the case project in were successful. As a result, the infrastructure as code shortened approximately by 2400 lines, which is not an insignificant amount. The infrastructure as code is now cleaner and follows common programming practices better, from both security and code quality perspective. The resulting infrastructure remained relatively the same as in the beginning, with only notable changes being the handling of IAM privileges and the universal deployment trigger.

Refactoring the modules to be more universal reduced the amount of repeated code, as some near-identical modules are no longer declared multiple times. The modules now follow single responsibility principle better, while still being truly reusable. Creating new resources with the improved modules is effortless, as the modules handle creating the necessary resources with sensible configurations. However, refactoring the modules creating Cloud Build triggers was not possible in the way that was planned, which resulted in leaving those modules untouched.

Introducing the stack of modules approach was the biggest single contributor to reducing the amount of the code. The fundamental IaC architecture change eliminated thousands of lines copied and pasted Terraform code, resulting in much leaner infrastructure as code. As a result, most declarations and references need to be made only once, to the stack, instead of previous three times, once to each environment. This greatly re-

duces error-prone manual work and makes the infrastructure as code less cumbersome. Modifications and updates to the infrastructure are not only easier for the developers to make, but also include fewer vectors for bugs. The process of introducing the stack to a production-use infrastructure was successful and progressed according to the plan. Some actions needed to be taken in order to ensure that no data is lost, but despite the manual work there were no big blockers when introducing the changes. The development team of the case project considers this a sizable improvement to the infrastructure as code, and expects the stack to positively affect the maintaining and developing of the infrastructure.

Removing the Terraform generated secret values simplified the process of managing secret values, as all secret values are now managed by hand. This was the preference of the development team, as most of the secrets are keys and access tokens to external services, which need to be managed by hand in any case. The selected approach requires the developers to generate some secret values with the tool of their choice, but are no longer restricted to the `random` provider. However, this thesis did not improve the process of rotating secret values, as for example the database passwords need to be updated separately in the database. Rotating secret values also requires redeploying the containers using the updated secrets.

Implementing the universal deployment trigger greatly shortened the deployment process to staging and production environments. Before, in order to deploy images to all Cloud Run instances, 14 triggers needed to be manually executed by first searching the correct triggers and the pasting the correct commit hash. Now, this can be done by executing just one trigger using the commit hash. The deployment trigger is also distinctively named, so that is easy to find and is not mistaken for some else trigger. The universal deployment trigger shortened the deployment process from five to ten minutes of error-prone work to merely seconds, with many fewer steps.

As a result of properly setting version constraints and refining the granting of IAM roles, the case project now better follows good practices and conventions. The version constraints to Terraform, Google provider and third-party modules improve the maintainability of the infrastructure as code and aid the updating of said blocks. Restructuring the way IAM roles are granted makes the infrastructure to comply with the principle of least privileges better.

One of the original action points, action point 7, remained unsolved. Based on the research, no single best solution for Cloud Run instances to access their own URLs was found, but multiple subpar ones. However, as the case project is already quite deep in its lifecycle, a shortcut to defining the URLs was discovered. The URL for a new Cloud Run instance can be deduced from the URLs of existing Cloud Run instances, allowing it to be hard-coded. This will help the possible future cases where the need to pass a URL of

a Cloud Run instance to itself rises.

7.2 Answers to the research questions

Two research questions were set for this thesis in Chapter 4. The goal was to answer these questions through the case project, implementing the action points defined in Chapter 5. Working on the action points partially answered the research questions, although no silver bullets were discovered.

First research question asked *How to improve maintainability and developer experience of IaC?* Based on the results of this thesis, the same rules of law apply to infrastructure as code as to all other programming. When developing infrastructure as code, developers should strive to create simple and reusable code, which is often encouraged in other types of programming too. Learning the tools and their quirks properly will also help in producing maintainable code, as the tools at hand are correctly utilized. The single biggest improvement to maintainability and developer experience is the stack of modules architecture change, which shortened the infrastructure as code significantly and reduced the amount of manual work done by developers when developing the infrastructure.

The second research question was *How to make fundamental changes to IaC after going to production?* Based on the improvements done to the case project in this thesis, there is nothing magical to making even big changes to infrastructure as code in production. The biggest challenge in changing IaC in production, especially when working with Terraform, is the tendency of Terraform to recreate resources when their declaration fundamentally changes, even though the end result stays the same. This may lead to data loss, if not addressed properly. Making changes to production infrastructure as code required through planning and executing in order not to lose any data or cause unnecessary disruptions to the service, but other than that required nothing out of the ordinary.

7.3 Best practices

This section presents a set of best practices to consider when writing infrastructure as code, based on the results of this thesis and the subjective experiences of working on the case project for approximately a year. These items are all the more important to consider when starting a fresh project as they can help to avoid the same pitfalls the case project encountered. However, the points are also relevant for ongoing projects and as demonstrated in this thesis, it is possible to make the required changes even when the infrastructure is already in production.

Use version constraints. Terraform along its modules and providers, are software just like other programming languages, frameworks, libraries and run-times. Version constraints should be set to lock down specific versions of the

software and gain deliberate control of updating them. This reduces the risk of accidentally breaking working infrastructures with unplanned updates and gives detailed information of the software in use.

Create a seed Terraform. When first starting out an empty project, consider creating a seed Terraform to initialize the project for GitOps process. This is all the more important when working with multiple, near-identical environments, which live in separate Google Cloud projects. The seed project enables APIs and creates resources required for the GitOps process to work. Creating a seed eases the process of creating multiple environments, as they can be initialized using the seed. The seed also doubles as a record of which APIs and products are enabled, as they are controlled using IaC instead of manually clicking through the graphical user interface. However, the seed should only be used for resources which need higher privileges, such as enabling APIs or are required for the GitOps process to work, such as setting up service accounts.

Use the stack of modules approach. When working with multiple, almost identical environments, consider using the stack of modules approach presented in this thesis. The stack of modules approach greatly decreases the amount of Terraform code, and therefore makes maintaining and further developing the IaC code much less manual and error-prone. The stack of modules approach allows the main content of an infrastructure to be declared only once, instead of declaring every resource, binding, reference and linking multiple times, once to each environment. Once the stack of modules is implemented, introducing new environments is easy, and new resources are instantly in use in all environments with just modifying the stack. The stack of modules eliminates much of the manual, repetitive work of maintaining multiple environments of the same system.

Utilize Terraform to its fullest. Instead of simply declaring resources using a provider and calling it a day, make an effort to learn and utilize Terraform and its constructs. Terraform contains many of the commonplace programming language constructs, such as variables, loops and conditional expressions. Utilize the tools available to produce modular, reusable IaC code instead of hard-coding everything. Build modules smartly, so that they are truly universal and abstract unneeded details away but are not hard to use and do not get in the way of implementing new features.

Modify production infrastructure carefully. When introducing changes to a production-use infrastructure, research and plan the changes thoroughly. Remember to utilize transition periods, i.e. parallel queues, and intermediate resources to avoid losing any data or causing disruptions to the service. Be sure to not accidentally delete any data when making changes that may result in destroying or recreating the resources. Utilize a service break if need be.

Obey the principle of least privilege. The principle of least privilege is a very common security principle applying to all sectors of IT. When granting privileges using Terraform, use correct resources which only grant privileges on the lowest needed level. For example, if a Cloud Run instance A needs to call Cloud Run instance B, give the instance A privileges to call instance B only, using `google_cloud_run_service_iam_member` resource. Avoid using project-wide IAM resources.

Decide how to control secret values. Generating secret values with Terraform may not be the best solution, although possible. Decide on the approach to secret values early on and follow it. Terraform provides tools to generate and rotate values, but the rest of the infrastructure needs work along with it. In the case project, secrets are created using Terraform, but their values are managed by hand and are not part of the infrastructure as code declaration.

Consider not using Cloud Build for CI/CD. Consider if implementing the CI/CD pipeline using some other tool than Google Cloud Build is feasible. Although Cloud Build gets the job done, there might be better alternatives, such as GitHub Actions. In the case project there are more than 70 triggers for building and deploying 11 images to 14 Cloud Runs, which makes working with them exhausting. If nevertheless working with Cloud Build, implement automatic triggers handling building and deployment. If deployment needs to be behind a manual trigger, implement a multistep trigger deploying all the images at once.

Infrastructure as code is still code. Remember that infrastructure as code is code as much as code written in other languages. It should follow the same rules of law and practices than all other programming. Strive to produce clean, easy-to-understand and reusable infrastructure as code. Follow common programming practices and principles, such as *don't repeat yourself (DRY)*, *keep it simple, stupid (KISS)*, single responsibility, separation of concerns, and *you aren't going to need it (YAGNI)*. The principles introduced are based on [31].

8. CONCLUSION

This thesis presents the process of improving the maintainability and developer experience of an infrastructure as code implementation for a production-use case project. At the beginning, related concepts and tools were introduced, and the case project was presented. The shortcomings of the case project were then discussed, and an action plan was formed. The plan was implemented and evaluated, and a set of best practices was compiled.

The implementation plan focused on improving previously identified problems in the case project infrastructure as code implementation, divided into seven discrete action points. Overall, the implementing of the action points was successful. Most of the action points were implemented as planned, except action point 7, which was not possible to implement due to technological constraints. The implementation of the action points improved the infrastructure as code greatly, shortening the implementation by some 2400 lines, reducing the need for tedious manual work and making the implementation to comply with good practices better. Some implemented changes are already in production, and some still require work before they can be deployed. The case project was successfully improved and now better allows future development and maintenance.

The research questions were partly answered. Improving the maintainability and developer experience of infrastructure as code is similar than with all other code. Developers should strive to create modular, clean and simple code, as is true with other types of programming. The thesis also discovered the stack of module approach of organizing infrastructure as code, which was the single biggest factor in improving the case project infrastructure as code. Making fundamental changes to production use infrastructure as code proved to be rather trivial, requiring nothing more than careful planning and research of the change at hand. Although some answers were found, this thesis did not discover any significant new contributions, but rather reaffirmed existing practices and conventions.

In addition, a set of best practices was compiled based on the results. The best practices included general suggestions for using infrastructure as code as a whole, as well as more specific suggestions related to working with Terraform and Google Cloud in particular. The set of best practices was compiled with the goal of avoiding the pitfalls of the case project in the future.

Future work could focus on the measurement of maintainability regarding infrastructure as code, as this thesis did not scientifically measure the maintainability. A comprehensive survey could be conducted to find out more about the developer experience related to working with infrastructure as code. The best practices could also be expanded and more thoroughly verified, other than with just the case project of this thesis. After all, they are now based on the subjective experiences and observations of the author.

REFERENCES

- [1] Alde Alanda, Hanriyawan Adnan Mooduto, and Rizka Hadelina. “Continuous Integration and Continuous Deployment (CI/CD) for Web Applications on Cloud Infrastructures”. In: *JITCE (Journal of information technology and computer engineering) (Online)* 6.2 (2022). ISSN: 2599-1663.
- [2] *ALTER ROLE. PostgreSQL 8.0.26 Documentation*. URL: <https://www.postgresql.org/docs/current/sql-alterrole.html> (visited on 05/22/2023).
- [3] Matej Artac et al. “DevOps: Introducing Infrastructure-as-Code”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 497–498. DOI: 10.1109/ICSE-C.2017.162.
- [4] Florian Beetz, Anja Kammer, and Dr. Harrer Simon. *GitOps – Cloud-native Continuous Deployment*. innoQ, 2021.
- [5] *Best practices for using Terraform. Google Cloud*. 2023. URL: <https://cloud.google.com/docs/terraform/best-practices-for-terraform> (visited on 02/14/2023).
- [6] Yevgeniy Birkman. *Terraform: up and running: writing infrastructure as code*. O’Reilly, 2019.
- [7] Ian Buchanan. *Infrastructure as code. Atlassian*. URL: <https://www.atlassian.com/microservices/cloud-computing/infrastructure-as-code> (visited on 06/05/2023).
- [8] Celia Chen et al. “Why Is It Important to Measure Maintainability and What Are the Best Ways to Do It?” In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 377–378. DOI: 10.1109/ICSE-C.2017.75.
- [9] *Cloud Run IAM Roles. Google Cloud*. URL: <https://cloud.google.com/run/docs/reference/iam/roles> (visited on 05/11/2023).
- [10] *Cloud Scheduler. Google Cloud*. URL: <https://cloud.google.com/scheduler> (visited on 06/10/2023).
- [11] *Cloud SQL. Google Cloud*. URL: <https://cloud.google.com/sql> (visited on 06/10/2023).
- [12] *Cloud Storage. Google Cloud*. URL: <https://cloud.google.com/storage> (visited on 06/10/2023).
- [13] *Configure Cloud SQL and the external server for replication. Google Cloud*. URL: <https://cloud.google.com/sql/docs/postgres/replication/configure-replication-from-external> (visited on 04/13/2023).
- [14] *Configure cron job schedules. Google Cloud*. URL: <https://cloud.google.com/scheduler/docs/configuring/cron-job-schedules> (visited on 05/13/2023).

- [15] *Configuring the order of build steps*. Google Cloud. URL: <https://cloud.google.com/build/docs/configuring-builds/configure-build-step-order> (visited on 05/18/2023).
- [16] *Connect to a GitHub Repository*. Google Cloud. 2023. URL: https://cloud.google.com/build/docs/automating-builds/github/connect-repo-github?generation=1st-gen#connecting_a_github_repository (visited on 02/24/2023).
- [17] *Data Sources*. HashiCorp. URL: <https://developer.hashicorp.com/terraform/language/data-sources> (visited on 06/08/2023).
- [18] *Developer Experience (DevEx)*. Microsoft: *Code With Engineering Playbook*. URL: <https://microsoft.github.io/code-with-engineering-playbook/developer-experience/> (visited on 06/10/2023).
- [19] *DevOps*. Atlassian. URL: <https://www.atlassian.com/devops> (visited on 06/05/2023).
- [20] Jeff Doolittle and Robert Blumen. "Luke Hoban on Infrastructure as Code". In: *IEEE software* 39.2 (2022), pp. 112–114. ISSN: 0740-7459.
- [21] *dynamic Blocks*. HashiCorp. URL: <https://developer.hashicorp.com/terraform/language/expressions/dynamic-blocks> (visited on 05/16/2023).
- [22] Christof Ebert and Lorin Hochstein. "DevOps in Practice". In: *IEEE software* 40.1 (2023), pp. 29–36. ISSN: 0740-7459.
- [23] *Export and import using pg_dump and pg_restore*. Google. URL: <https://cloud.google.com/sql/docs/postgres/import-export/import-export-dmp> (visited on 04/13/2023).
- [24] *Export and import using SQL dump files*. Google Cloud. URL: <https://cloud.google.com/sql/docs/postgres/import-export/import-export-sql> (visited on 04/13/2023).
- [25] *Expressions*. HashiCorp. URL: <https://developer.hashicorp.com/terraform/language/expressions> (visited on 06/08/2023).
- [26] Martin Fowler. *Code Smell*. *martinfowler.com*. URL: <https://martinfowler.com/bliki/CodeSmell.html> (visited on 06/10/2023).
- [27] *Geography and regions*. Google Cloud. URL: <https://cloud.google.com/docs/geography-and-regions> (visited on 06/08/2023).
- [28] *git-diff - Show changes between commits, commit and working tree, etc*. Git. URL: <https://git-scm.com/docs/git-diff#Documentation/git-diff.txt-emgitdiffemltoptionsgt--no-index--ltpathgtltpathgt> (visited on 04/20/2023).
- [29] *Google Cloud's buildpacks*. Google Cloud. 2023. URL: <https://cloud.google.com/docs/buildpacks/overview> (visited on 02/28/2023).
- [30] *google_cloudbuild_trigger*. Terraform Registry. URL: https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/cloudbuild_trigger#wait_for (visited on 05/18/2023).
- [31] Anthony Grant. *10 Basic Programming Principles Every Programmer Must Know. Make Use Of*. URL: <https://www.makeuseof.com/tag/basic-programming-principles/> (visited on 06/01/2023).

- [32] Ted Hunter and Steven Porter. *Google Cloud Platform for developers: build highly scalable cloud solutions with the power of Google Cloud Platform*. Packt Publishing, 2018.
- [33] *IAM policy for Cloud Run Service*. *Terraform Registry*. URL: https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/cloud_run_service_iam (visited on 05/29/2023).
- [34] “IEEE Standard Glossary of Software Engineering Terminology”. In: *IEEE Std 610.12-1990* (1990). DOI: 10.1109/IEEESTD.1990.101064.
- [35] *Invoking with as HTTPS request*. *Google Cloud*. URL: <https://cloud.google.com/run/docs/triggering/https-request> (visited on 05/29/2023).
- [36] Sven Johann. “Kief Morris on Infrastructure as Code”. In: *IEEE software* 34.1 (2017), pp. 117–120. ISSN: 0740-7459.
- [37] *Kari Lukka: Konstruktiiivinen tutkimusote*. *Metodix*. URL: <https://metodix.fi/2014/05/19/lukka-konstruktiiivinen-tutkimusote/> (visited on 04/08/2023).
- [38] Eero Kasanen, Kari Lukka, and Arto Siitonen. “The constructive approach in management accounting research”. In: *Journal of management accounting research* 5 (1993), pp. 243–. ISSN: 1049-2127.
- [39] Mikael Krief. *Terraform cookbook: efficiently define, launch, and manage infrastructure as code across various cloud platforms*. Packt Publishing, 2020.
- [40] Kari Lukka. “The key issues of applying the constructive approach to field research”. In: *Management Expertise for the New Millennium* (Jan. 2000), pp. 113–128.
- [41] *Manage Terraform Versions*. *HashiCorp*. URL: <https://developer.hashicorp.com/terraform/tutorials/configuration-language/versions> (visited on 05/23/2023).
- [42] *Managing infrastructure as code with Terraform, Cloud Build, and GitOps*. *Google Cloud*. 2023. URL: <https://cloud.google.com/docs/terraform/resource-management/managing-infrastructure-as-code> (visited on 02/14/2023).
- [43] Ravi Misha. *HashiCorp Infrastructure Automation Certification Guide: Pass the Terraform Associate Exam and Manage IaC to Scale Across AWS, Azure, and Google Cloud*. Packt Publishing, 2021.
- [44] *Modules*. *HashiCorp*. URL: <https://developer.hashicorp.com/terraform/language/modules> (visited on 06/08/2023).
- [45] *Move and rename buckets*. *Google Cloud*. URL: <https://cloud.google.com/storage/docs/moving-buckets#move-buckets-gcloud> (visited on 04/13/2023).
- [46] *Providers*. *HashiCorp*. URL: <https://developer.hashicorp.com/terraform/language/providers> (visited on 06/08/2023).
- [47] *psql*. *PostgreSQL Client Applications*. URL: <https://www.postgresql.org/docs/current/app-psql.html> (visited on 05/22/2023).
- [48] *Random Provider: Resource "keepers"*. *Terraform Registry*. URL: <https://registry.terraform.io/providers/hashicorp/random/latest/docs#resource-keepers> (visited on 03/10/2023).

- [49] Bob Reselman and Matthew Broberg. *Developer experience: an essential aspect of enterprise architecture*. Redhat. URL: <https://www.redhat.com/architect/developer-experience> (visited on 06/10/2023).
- [50] *Resource Behaviour*. HashiCorp. URL: <https://developer.hashicorp.com/terraform/language/resources/behavior> (visited on 06/08/2023).
- [51] *Resource hierarchy*. Google Cloud. 2023. URL: <https://cloud.google.com/docs/terraform/resource-management/managing-infrastructure-as-code> (visited on 02/14/2023).
- [52] *Resources*. HashiCorp. URL: <https://developer.hashicorp.com/terraform/language/resources> (visited on 06/08/2023).
- [53] Alexis Richardson. *GitOps - Operations by Pull Request*. weaveworks. URL: <https://www.weave.works/blog/gitops-operations-by-pull-request> (visited on 06/06/2023).
- [54] *Role operations*. node-pg-migrate. URL: https://salsita.github.io/node-pg-migrate/#/roles?id=pgmalterrole-role_name-role_options- (visited on 05/22/2023).
- [55] *Secret Manager Best practices*. Google. URL: <https://cloud.google.com/secret-manager/docs/best-practices> (visited on 03/17/2023).
- [56] *Security and Privacy Controls for Information Systems and Organizations*. Tech. rep. National Institute of Standards and Technology, 2020. URL: <https://doi.org/10.6028/NIST.SP.800-53r5>.
- [57] Vitthal Srinivasan, Janani Ravi, and Judy Raj. *Google cloud platform for architects: design and manage powerful cloud solutions*. Packt Publishing, 2018.
- [58] *String Templates*. HashiCorp. URL: <https://developer.hashicorp.com/terraform/language/expressions/strings#string-templates> (visited on 05/22/2023).
- [59] *Substituting variable values*. Google Cloud. URL: <https://cloud.google.com/build/docs/configuring-builds/substitute-variable-values> (visited on 05/22/2023).
- [60] *Terraform Settings*. HashiCorp. URL: <https://developer.hashicorp.com/terraform/language/settings> (visited on 04/05/2023).
- [61] *The Core Terraform Workflow*. HashiCorp. URL: <https://developer.hashicorp.com/terraform/intro/core-workflow> (visited on 06/08/2023).
- [62] *The for_each Meta-Argument*. HashiCorp. URL: https://developer.hashicorp.com/terraform/language/meta-arguments/for_each (visited on 05/13/2023).
- [63] *The Story of HashiCorp Terraform with Mitchell Hashimoto*. HashiCorp. URL: <https://www.hashicorp.com/resources/the-story-of-hashicorp-terraform-with-mitchell-hashimoto> (visited on 06/08/2023).
- [64] *Transfer between Cloud Storage buckets*. Google Cloud. URL: <https://cloud.google.com/storage-transfer/docs/cloud-storage-to-cloud-storage> (visited on 04/13/2023).
- [65] *Upgrading to Terraform v1.5*. HashiCorp. URL: <https://developer.hashicorp.com/terraform/language/upgrade-guides> (visited on 06/20/2023).
- [66] *Use secrets*. Google Cloud. URL: <https://cloud.google.com/run/docs/configuring-secrets> (visited on 05/23/2023).

- [67] Rosemary Wang. *Infrastructure as Code, Patterns and Practices*. Manning Publications, 2022.
- [68] *What is Cloud Run? Google Cloud*. URL: <https://cloud.google.com/run/docs/overview/what-is-cloud-run> (visited on 06/10/2023).
- [69] *What is DevOps? GitLab*. URL: <https://about.gitlab.com/topics/devops/> (visited on 06/05/2023).
- [70] *What is DevOps? Amazon Web Services*. URL: <https://aws.amazon.com/devops/what-is-devops/> (visited on 06/05/2023).
- [71] *What is Pub/Sub? Google Cloud*. URL: <https://cloud.google.com/pubsub/docs/overview> (visited on 06/10/2023).
- [72] *What is Terraform? HashiCorp*. URL: <https://developer.hashicorp.com/terraform/intro> (visited on 06/08/2023).
- [73] Liming Zhu, Len Bass, and George Champlin-Scharff. “DevOps and Its Practices”. In: *IEEE software* 33.3 (2016), pp. 32–34. ISSN: 0740-7459.