Bachelor’s Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

# Integration of DevOps for Software Development and Delivery

Vít Lupínek

Supervisor: Ing. Karel Frajták, Ph.D.
Field of study: Open Informatics
May 2022

## Acknowledgements

I want to thank my supervisor Ing. Karel Frajták, Ph.D. for valuable feedback during writing of this thesis. I also want to thank my family and friends for support.

## Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 19, 2022

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Praha, 19. května 2022

## Abstract

This thesis focuses on solving the challenges in development of cloud based applications following the microservice architecture. We first analyse these challenges, and we show ways to solve them with automation. We also describe the principles of Infrastructure as a Code and Continuous Integration and Delivery. Then we design and implement an easy to use system for small to medium sized development teams, which automates the setup of infrastructure for their application and also automates the process of building, testing and deploying new versions of the software. We put emphasis on simple use, readability, reusability while eliminating as much manual interference as possible, while keeping the ability to customise every aspect of the system.

**Keywords:** DevOps, infrastructure, IaC, infrastructure as a code, ci/cd, pipeline, continuous integration, continuous delivery, Terraform, Terragrunt, GitOps

**Supervisor:** Ing. Karel Frajták, Ph.D.

## Abstrakt

Tato práce se soustředí na řešení problému spojených s vývojem cloudových aplikací podle architektury mikroslužeb. Nejdříve tyto problémy analyzuje a ukážeme, jak je lze automatizovat. Také popisujeme principy Infrastruktury jako kód a Kontinuální integrace a nasazení. Dále v práci nejdříve navrhneme, a pak také implementujeme snadno použitelný systém pro malé a středně velké týmy, který automatizuje proces vystavění infrastruktury jejich aplikace a také automatizuje proces sestavení, testování a nasazení nové verze tohoto softwaru. Kladli jsme důraz na jednoduchost využití systému, čitelnost a přepoužitelnost a zároveň jsme chtěli eliminovat co nejvíc manuálních zásahů do konfigurace, při zachování možnosti vlastního přizpůsobení všech aspektů našeho systému.

**Klíčová slova:** DevOps, infrastruktura, IaC, infrastruktura jako kód, ci/cd, pipeline, kontinuální integrace, kontinuální nasazení, Terraform, Terragrunt, GitOps

**Překlad názvu:** Integrace DevOps ve vývoji a řízení softwarových řešení

# Contents

# BACHELOR‘S THESIS ASSIGNMENT

## I. Personal and study details

| | | | |
|---|---|---|---|
| Student's name: | **Lupínek Vít** | Personal ID number: | **483757** |
| Faculty / Institute: | **Faculty of Electrical Engineering** | | |
| Department / Institute: | **Department of Computer Science** | | |
| Study program: | **Open Informatics** | | |
| Specialisation: | **Software** | | |

## II. Bachelor’s thesis details

Bachelor’s thesis title in English:

**Integration of DevOps for Software Development and Delivery**

Bachelor’s thesis title in Czech:

**Integrace DevOps ve vývoji a řízení softwarových řešení**

Guidelines:

Design and implement a general infrastructure as a code for simpler management, development and testing of containerized applications. The system will allow automatic code linting and complete unit testing of the application, as well as end to end testing support for front-end web application. It will also automatically containerize the solution and ensure its smooth deployment to a chosen environment. The system will manage this environment through a set of text configuration files. Test functionality of the system on a simple example project.

Bibliography / sources:

1)The Devops Handbook, Authors: Gene Kim, Jez Humble, and Patrick Debois
2) Starting and Scaling DevOps in the Enterprise, Author: Gary Gruve

Name and workplace of bachelor’s thesis supervisor:

**Ing. Karel Frajták, Ph.D. System Testing IntelLigent Lab FEE**

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **27.05.2021** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **19.02.2023**

| | | |
|---|---|---|
| Ing. Karel Frajták, Ph.D. | | prof. Mgr. Petr Páta, Ph.D. |
| Supervisor’s signature | Head of department’s signature | Dean’s signature |

## III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

# Chapter 1
# Introduction

DevOps is a set of practices which aim to combine and simplify the work of developers (Dev) and IT operations (Ops). The main goal of DevOps is to create a more efficient process of developing software. There has not been developed a universal definition for the term "DevOps", so it is mostly defined by a set of key principles being: shared ownership, workflow automation, and rapid feedback[1]. Nowadays, programmers and software developers first encounter DevOps in the workflow automation form.

The number one priority for programmers is the actual code they write, if it runs smoothly, and solves the given problem efficiently. They do not want to worry about managing servers where their code will run, or configure data centres, or how the final application will be served to the end user. This theses focuses on solving most of these everyday problems for programmers through automation, and letting them focus on fast code development, and frequent updates to their applications.

## 1.1 Objectives

The main goal of this thesis is to implement an easy to use system of configuration files, that helps small teams provision infrastructure for a microservice architecture, and automate the process of testing, building, and deploying small services to the cloud. It also aims to allow for simple customisation of this process. Implementing such tool eliminates the need for manual adjustments to the infrastructure of applications thus removing most of the errors introduced by a human, and also increasing the team's efficiency of developing their product.

First, we will look at problems in development and application deployment teams usually come across. We will focus on those, that can be solved with automation, and what principles can be applied to eliminate these challenges. Then we will analyse existing solutions and talk about the implementation process of our system. We will also discus technologies, libraries, and third party applications which were used, and why we chose them.

The implementation will consist of configuration files that provide infras-

[1] Mentioned in [Lou12]

tructure provisioning for a cloud based web application following microservice architecture. There will also be an option to create multiple environments such as development, staging or testing, and of course a production environment. This system tries to create very similar environments to the production one with only small changes to for example database storage, number of instances for each microservice or the verbosity of log files. Last but not least the system will provide a fully functioning template CI/CD pipeline for linting, building, testing, and deployment of each microservice. We list the main goals below for clarity:

- out of the box provisioning for small to medium size projects
- simple extension of CI/CD pipelines
- ability to create environments on demand
- easy customisation of each step

## 1.2 Thesis Outline.

We briefly introduce each chapter of the thesis:

- *DevOps (as a Service)* - In Chapter 2, we introduce principles of DevOps, that can be automated through code, then we discuss challenges that developers encounter when developing new applications. We also discus the principles of Infrastructure as a Code(IaC) in more detail.
- *Technology Stack* - In Chapter 3, we present the technologies that were used in the implementation and why.
- *System Design* - In Chapter 4, we first visualise the design for our system, separate the system into smaller components, which we then describe in more detail.
- *Implementation Process* - In Chapter 5, we show the process of implementing this system, and discuss some of the challenges we faced in development. The implementation is split into two parts: IaC implementation and CI/CD implementation.
- *Example Project* - In Chapter 6, we test the system on an example E-commerce project, and provide some screenshots of this project as well as of the infrastructure our system has provisioned.

# Chapter 2
# DevOps (as a Service)

In this chapter, we will look at some of the problems that appear the most while developing a cloud native web service or application. As DevOps is a large set of practices, we will only focus on two parts: Infrastructure as a Code and Continuous Integration and Continuous Delivery as they work closely with the actual code of application and they can be implemented programmatically. We discuss some of the major benefits in section 2.2.6. This programmatic approach to DevOps is emerging as a new kind of philosophy, at least in the cloud environment, the DevOps as a Service philosophy. We will not use this term as it does not have a specific definition, however it describes our implemented system or the motivation behind creating it in a compact way.

## 2.1 Challenges to Solve

Let's first look at some of the challenges that need to be solved when developing and running a cloud native application following a microservice architecture.

### 2.1.1 Networking

In a microservice architecture networking is no longer as straight forward as it is with monolithic application. What used to be one simple function call to a remote service now requires a network hop, because we need to for example route our requests to different services, balance out the load and more. Distributing load among microservices and using virtual networks to isolate and speed up communication to them are challenges that we want to address with our system.

There are two main categories to keep in mind. Traffic within our networks e.g. communication among components of our data centre such as microservices, routers, or data storage. This is sometimes called East-West Traffic. The second category is managing traffic between our networks and the outer world e.g. from users and their browsers, or third party services. This type of communication is called North-South Traffic.

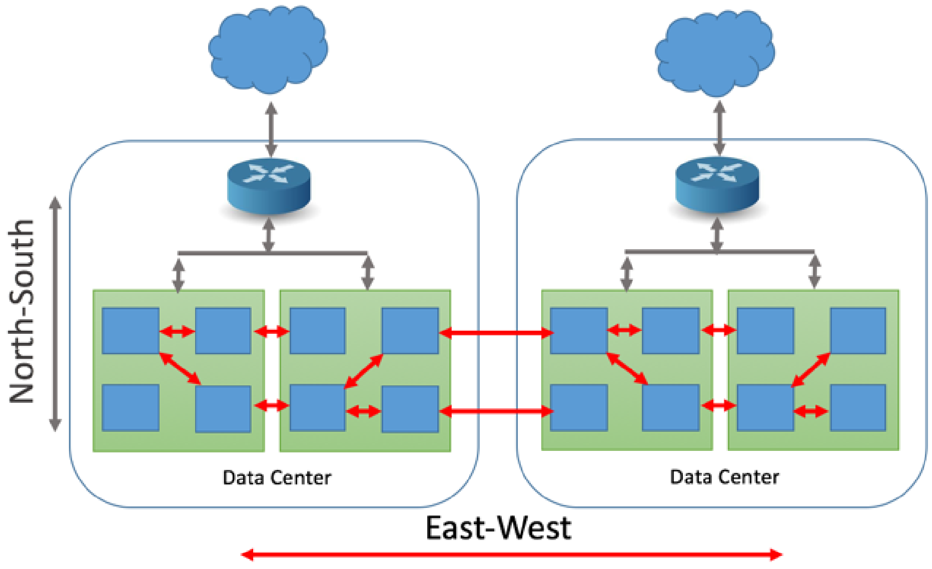


**Figure 2.1:** Visualisation of north-south and east-west traffic[NEU16]

## East-West Traffic

In a microservice architecture each microservice has its delegated function, for example one could manage user authentication, while another could only work around creating, updating and deleting orders. This microservice would need to ask user management service, if a user is authorised to access a specific order. Now allowing these components of our data centre to talk to each other very quickly is essential for an efficient and fast response time.

Because the usage of virtual systems has grown extensively, and because organisations now prefer private cloud infrastructure more and more, east-west traffic volumes have increased drastically. Nowadays there are many functions and services performed virtually, instead of how they used to – on physical hardware. We need to treat these services as a dynamically updating software. We need to update the configuration whenever we create or destroy new microservices in our system, or when we want to change how to balance out the load. We also need to keep in mind that parts of our system can crash and for example when we do not want to route requests to a crashed service instance the load balancer configuration needs to be updated accordingly.

**Virtual Private Cloud.** In the cloud computing environment servers and other resources can share physical networking resources with other people's servers. To isolate data centres and networks, cloud service providers introduce the ability of creating a Virtual Private Cloud which acts as a logical isolation solution that gives us control over networking environment the same way as if we ran infrastructure on-premise.[1]

This abstraction provides us with one very important advantage that

[1]Meaning the same way as if we had all the servers run on our own hardware in a dedicated place only we could access.

is the ability to communicate among services over a local network, which greatly decreases the latency and also isolates their communication from the outer world.

### North-South Traffic

Any communication between components of a data centre and another system, which is physically out of the boundary of the data centre, is referred to as north-south traffic. For instance a client requesting access to web application.

Traffic coming into the data centre through a firewall or other network device is referred to as southbound traffic. The opposite of it, traffic going out of the data centre is referred to as northbound. This type of traffic is prone to attacks from the outside as it flows over the public internet and in some sense anyone can intercept it. Proper security control needs to be in place to ensure safe data transfer in and out of our data centres. There are plenty of tools to help us secure our infrastructure like Application Gateway, Firewall, Network ACL[2] and many more.

**Authorisation.** In an on-premise infrastructure there were some specific attacks for example malicious Office macros, and Powershell deployments, sometimes even phishing attacks. These issues are more or less irrelevant in the cloud environment as the cloud platforms usually provide protection against such attacks, however new risks occur when using cloud as the attack surface increases. One major threat is access to individual resources in the infrastructure. A big part of North-South traffic, and to some degree also East-West traffic, is deciding on who has access to our data centre components e.g. who is authorised to use the resource. This is usually handled via firewall or a virtual abstraction of it such as security groups in AWS. They control which IP addresses, ports or protocols can be accessed in our data centre.

### Load balancing

Balancing load among not only different microservices, but also splitting work among multiple instances of these services is a key point in our system. We can imagine 2.2 Load Balancers as routing and naming devices, that redirect network traffic to individual components of our data centre. These devices are able to check the health of our application instances and not send traffic to servers that are unavailable or route that work to different available instances in the system.

[2]Access Control List

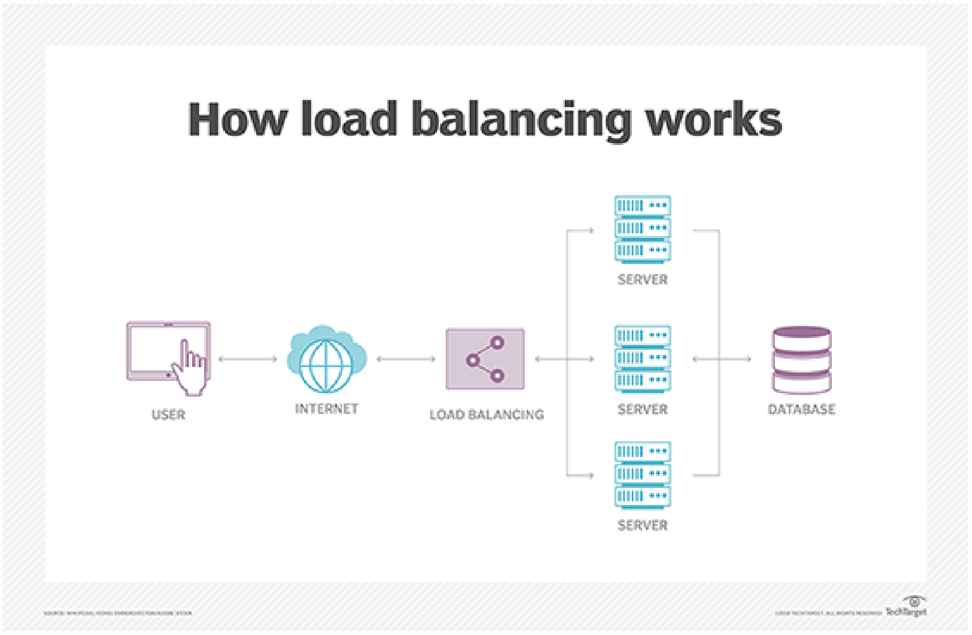


**Figure 2.2:** How Load Balancers work [Awa]

Load balancers are able to leverage algorithms for splitting work including:

- **Round Robin** - As the name suggest round robin strategy distributes network load around instances in turns. It forwards the first request to the first instance, the second request to the next instance and so on, until it runs out of instances to forward and then it starts again with the first instance. This is a very simple way of distributing work and is usually outperformed by the other approaches.
- **Least connection** - The least connections algorithm stores a record of active connections and forwards new connection to the server with the least number of active connections. [3]

Balancing load can be done at different layers of the ISO/OSI model. We can have an *Application Load Balancer* managing network traffic at the application level (HTTP/HTTPS) and make routing decisions based on metadata in requests. For example the AWS ALB supports making decisions according to different paths, sub-domains or header contents in each request.

*Network Load Balancers* on the other hand handle load at the network layer (TCP/UDP) and they do not analyse the contents of each request but only forward traffic.

## 2.1.2 Scalability

Scalability describes a system's elasticity. While we often use it to refer to a system's ability to grow, it is not exclusive to this definition. We can scale

[3]There is also a Weighted version of this algorithm which only adds weights according to priority of each instance

down, scale up, and scale out accordingly.

When we run a software product e.g. a website, web service, or application[4], we can measure a websites reach by the amount of network traffic it receives. For example in E-commerce more network traffic generally means a higher probability of selling goods, however it can bring other challenges we need to solve when ensuring quality interaction between our product and the visitor, such as delay when serving the application, or when we communicate with data storage. It is common to underestimate the amount of network traffic our website will incur, when designing it, or in the early stages of development. This can result in crashed servers and decline in quality of our product, which can lead to less people visiting our website or service.

In one sentence, scalability describes how our product adapts to change and demand. We talked about more people visiting our website, a higher demand, and providing more resources, but we also need to keep in mind, that adding more resources increases the operating costs and when the demand decreases we need to remove these newly added resources to cut back on expenses.

There are two types of scaling your application or its underlying infrastructure. *Vertical* and *Horizontal* scaling.

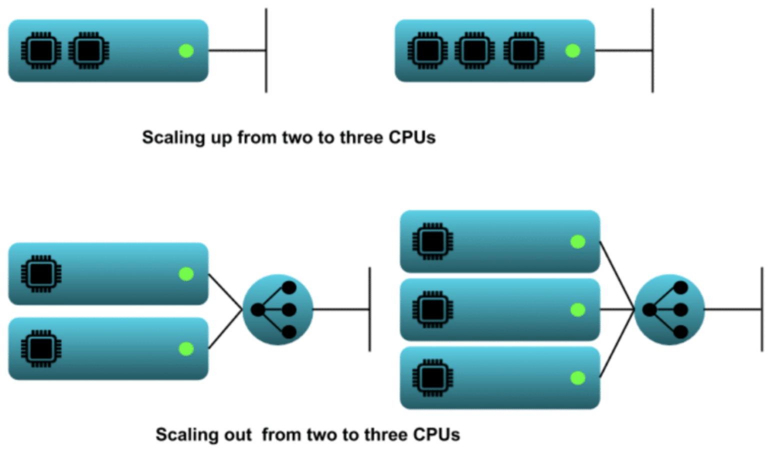

Scaling up from two to three CPUs
Scaling out from two to three CPUs

**Figure 2.3:** Difference between Vertical scaling (scaling up) and Horizontal scaling (scaling out)[Sta20]

## Vertical Scaling

*Vertical* scaling, or scaling up and down, means adding, or decreasing resources to maintain performance in changing demand. CPU, memory, network or storage are the most common targets for scaling up and down. It could also mean replacing a server with an upgraded one. At the same time scaling up

[4]We will use these terms interchangeably

could easily mean allowing our application to allocate more memory.

Let's say we have a simple client/server application where the client makes HTTP requests to our server application, a back end API[5]. As the time goes there are more and more end users using our application. The client side is run in a browser of a visitor so there is not an issue with performance. Every browser makes requests to the same server instance and we need to handle all of these requests on our server at the same time, but there is not enough processing power to respond to each request without a delay that would affect the end user's experience. With vertical scaling we could increase the CPU on the machine that hosts our server application, thus increasing the performance and hopefully respond to requests faster. For example in AWS terms if we had our application running on a t3.micro instance we could upgrade it to a t3.medium which would give us twice the processing power and also more memory. This approach has many disadvantages including:

- **Server downtime** - In order to increase resources on a server you usually need to stop the server and the application with it, then increase its resources, start the server up again and restart your application. While this process is going on, no users will be able to access your application. In an on-premise environment buying and replacing new components for our servers could take days or even weeks, but we will stick to a cloud based environment where this process usually takes minutes. Still this delay can lead to potential loss of profit. A big enough E-commerce application with downtime in minutes could lose hundreds of thousands of dollars in revenue. Also users might get angry, that the website is down and never come back to it.

- **Single point of failure** - Having all operations on a single server, increase the chance of losing all your data if a hardware or software failure occurs. In the event of a server failure your application crashes, and is unavailable for the end user which again leads to potential revenue loss.

- **Limited upgrade options** - There is an upper limit to how much you can upgrade a server. Every machine has its threshold for RAM, storage, and processing power.

Taking into account all of the above stated disadvantage for very small projects, or when you are on a tight budget, it still might be more beneficial to start with vertical scaling. Let us list the advantages bellow:

- **Cost effectiveness** - Upgrading an already existing machine is much cheaper then purchasing a new one. You are still running the same operating system, or using the same virtualisation.

- **Easier communication among services** - When you have only one instance of your application, for example a back end application communicating with only one database instance, there is no need to balance the

[5]Application Programming Interface

network traffic as it is always being handled by the same server. Also you have data in one place, there is no need to manage replication and synchronisation of data among storage nodes.

- **Simpler development of services** - Usually it is much simpler to develop an application, which runs on a single machine as developers do not have to think about their application being stateless, thus lowering the initial cost for developing a functioning application.

- **Easier maintenance** - Maintaining one instance is always going to be easier then managing multiple ones. You do not have to think about setting up new backups, or upgrading operating systems on multiple servers.

### Horizontal Scaling

On the other hand *Horizontal* scaling, or scaling out, refers to creating new instances of the same application. It could mean starting a new container with the same application on the same server and only balancing network load among these containers, or it could mean adding a whole new machine to our infrastructure. These new instances are all functionally exactly the same, in fact they do not have to know anything about the other running instances, or that they even exist. They also do not have to run on the same server or in the same country. That adds a new level of complexity for both the developers and DevOps engineers. We now need to keep in mind, that there is no guarantee which instance will process a given action and every instance has to handle it in the exact same way.

Going back to the example client/server application above, when facing a higher demand, scaling out could be accomplished by setting up a new server, installing the same software on this node and running our application. But then we would need to somehow decide on which instance handles an incoming request. This is called load balancing, we discussed different approaches to load balancing in section 2.1.1. After a load balancer is setup it is much simpler to add new nodes to this application cluster and redirect network traffic. In the world of AWS cloud computing we could achieve this by spinning up a new EC2 instance with an elastic IP address, then create an Application Load Balancer(ALB)2.1.1 which would distribute the network load between these two instances. The end users should not see a difference, they still access the same endpoint, only with this setup, there is no guarantee which EC2 instance will receive the request

To visualise this in figure 2.4 we illustrate a simple EC2 setup with a Internet Gateway to have it accessible from the internet.

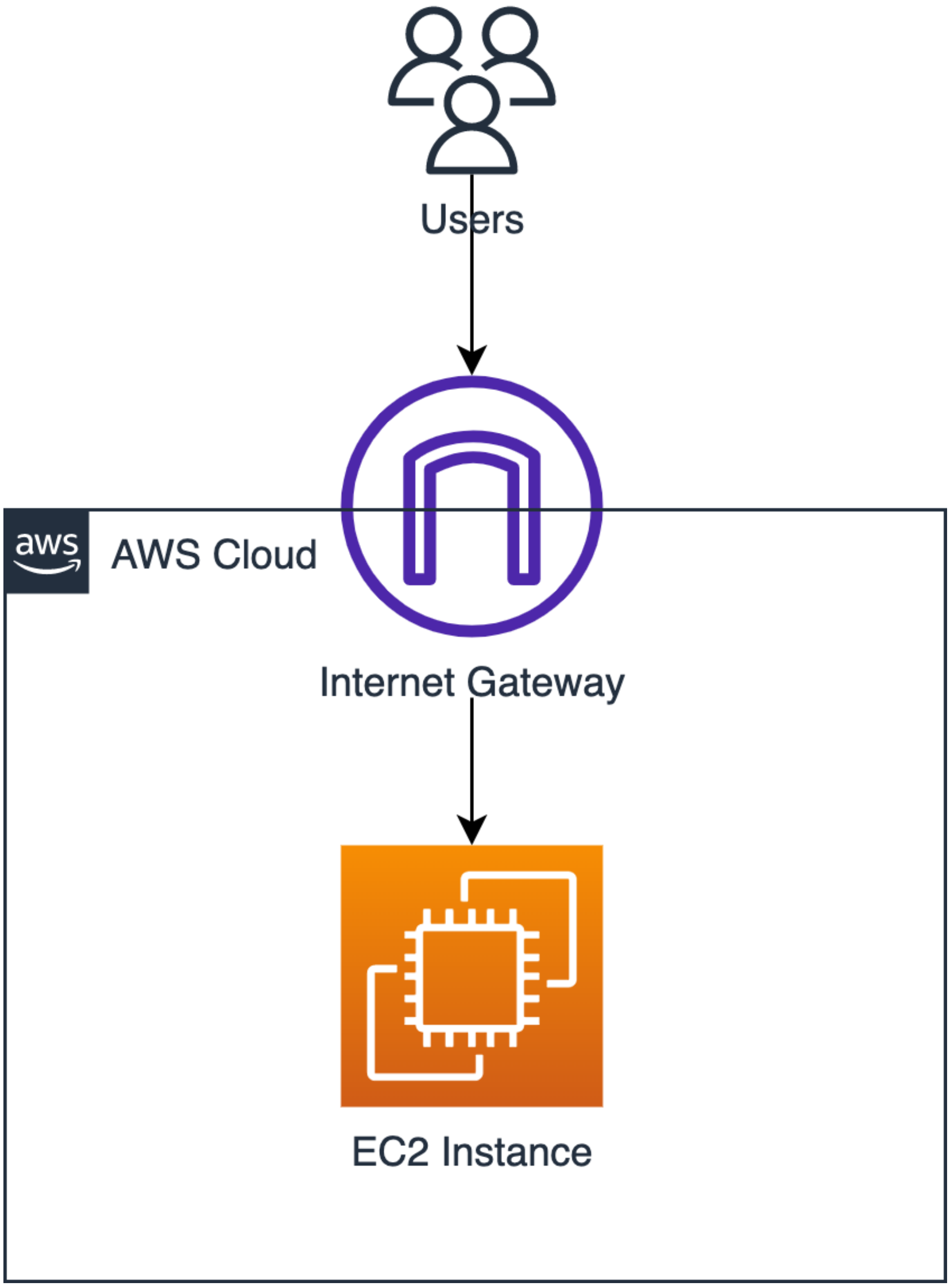


**Figure 2.4:** EC2 with Internet Gateweay setup

And then in figure 2.5 we show a possible setup with two EC2 instances, and an Application Load Balancer to distribute the network traffic.

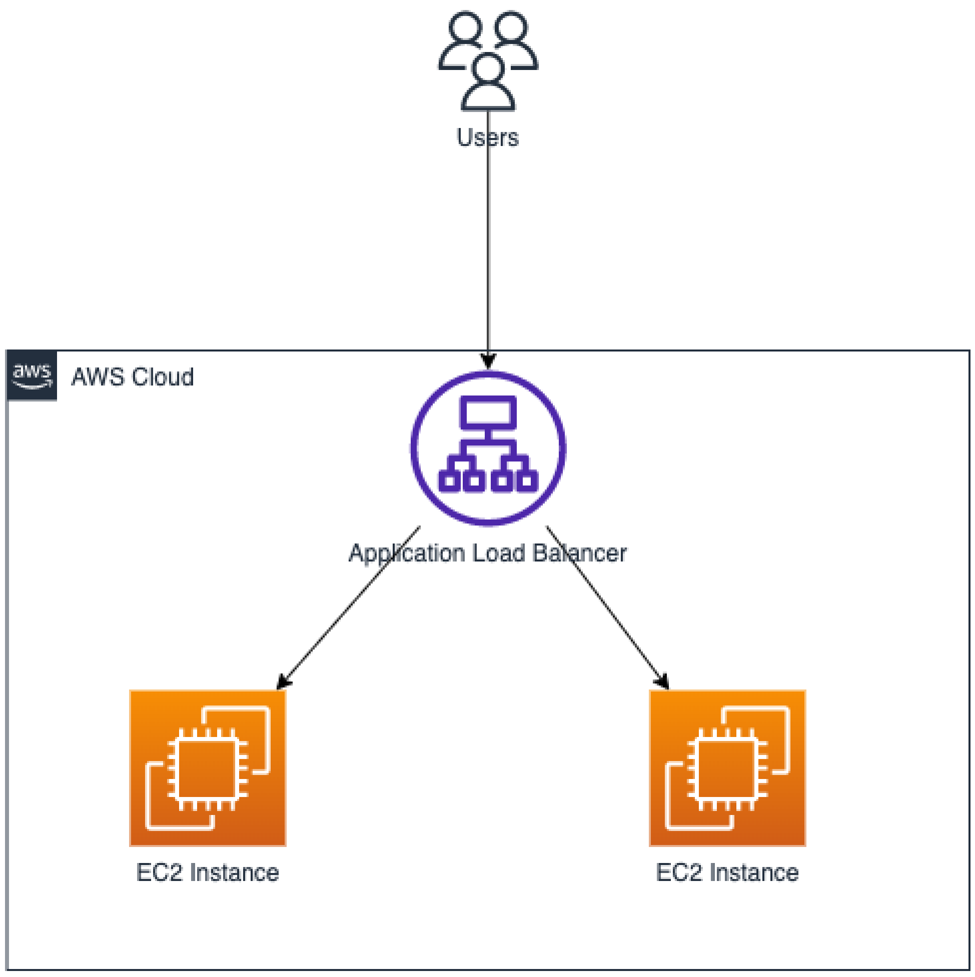


**Figure 2.5:** Two EC2s with ALB setup

One could argue scaling out requires a lot of steps and this is partially true. The complexity of such setup is one of the main disadvantages of horizontal scaling. Some of the disadvantages are listed bellow:

- **System complexity** - Adding load balancers and possibly virtualisation to your system, increases the possible points of failure in your system. Backing up your machines may also become a little more complex. You will need to ensure that nodes synchronise and communicate effectively. This possibly increases the delay in response to a user request.
- **Higher costs** - Multiple servers all need to run their operating systems and have their own storage and RAM, this is obviously more expensive, then keeping all data and operations on a single server.
- **More complicated development process** - Keeping with some if not all of the principles described in The Twelve-Factor App[6] is essential when creating an application that can be scaled out. This would require

[6]`https://12factor.net/`

more experienced developers, who generally demand higher salaries. Also designing a system with horizontal scaling in mind proves to be slightly more difficult then an application which would only scale vertically.

- **Maintenance** - More servers, load balancers, virtualisation, all of these need to be maintained which is again harder then maintaining a single server. Keeping everything exactly the same, upgrading all servers, deploying to all servers proves to be a difficult task, that usually can not and should not be done manually, but solved through automation tools. Again this brings another level of complexity to the project and possibly requires a specialist to set this process up.

On the other hand having multiple instances distributed around the world all behaving exactly the same brings a lot of benefits to a project. Let's again list them out:

- **Simpler and more elastic scaling** - From a hardware perspective, adding a new machine to your cluster is much simpler, than analysing what resource needs to be upgraded. Also spinning up a new instance, or tearing one down proves to be much more viable when dealing with spikes in demand. This is much more cost effective as your project grows in size and demand, but also if managed properly, scaling down when needed could save a small project money when additional nodes are not needed.

- **Absence of a single point of failure** - Having multiple instances of your application, or even better having multiple servers running the same application with distributed data among multiple locations eliminates the risk of losing all your data. If you have many servers with a copy of your application, when one instance crashes, another one can handle requests, meaning the end user never notices that one of your servers does not respond and could be on fire.

- **No or fewer periods of server downtime** - This closely relates to the statement above, when you have multiple instances of your application one being down does not affect the end user. We can use this when rolling out updates to our application by having an instance handle requests while we upgrade a second instance to a new version, then redirecting network traffic to this updated instance and updating the previous one. Some practices like blue-green deployment or rolling updates in Kubernetes take advantage of this benefit.

- **Possibly unlimited scaling** - Theoretically there is no upper limit to how many instances of your application can run at the same time as long as you have enough finances to cover the cost of the underlying infrastructure.

### Which Method To Use?

As always there is no clear answer to which method to use. Both horizontal and vertical scaling have their benefits and limitations. Also they are not mutually exclusive, so to some extend we can benefit from using both.
When deciding on which scaling option your organisation needs, you might want to consider some of these factors.

- **Cost** - As stated above, horizontal scaling has a higher initial cost ten vertical scaling, but could be potentially more cost effective long term.
- **Future of your project** - Keep in mind how much your project can grow. A project that will not need to be expanded by a lot would not benefit much from being able to horizontally scaled and scaling it up and down would be sufficient. On the other hand if you expect to grow not only in demand, but you will also need to serve your application in multiple countries around the world, then being able to spin up new instances, e.g. scaling out, that can be run anywhere in the world, will greatly simplify the process of expansion.
- **Reliability** - Horizontal scaling generally provides a more reliable system as it is more redundant then vertical scaling.
- **Complexity** - A simple straightforward application will not benefit much from being run on multiple machines, it could even decrease its performance.

### 2.1.3 Infrastructure Provisioning and Configuration

When creating a web based software such as a web application, or any other cloud application, we need to be able to create and setup the corresponding IT infrastructure. This process is handled by provisioning and configuration.

#### Provisioning vs Configuration

Provisioning is sometimes confused with configuration, but while both of them are commonly solved together in the development process, they are not the same thing. Provisioning focus more on the physical hardware or at least the virtualisation of such hardware. Once something is provisioned, only then it can be configured. That is where configuration steps in and solves the setup of software on a newly provisioned machine.

**Provisioning.** We wrote about Networking 2.1.1 and Scalability 2.1.2 as some of the more complex tasks that need attention while developing a new application. These fall mostly under the domains of provisioning[7] and because they are so complex, setting them up manually could introduce complications and possible errors made by us, humans. That is one of the reasons why we

[7]Although some networking issues for example proxiyng could be handled by tools such as NGINX, thus falling under the category of configuration of these tools.

want to automate the process of provisioning infrastructure for our applications.

There are tools that make it more convenient for us to manage provisioning automatically with text files for instance Terraform and its wrapper Terragrunt, but we will discuss them more in section 2.2.5

When the term "provisioning" is used it can mean different types of provisioning, for example:

- **Network Provisioning** - This can include setting up a network that the users, servers, containers, other services can access. In the hardware world it could mean setting up cables and wiring. But for us in the cloud it could mean creating a VPC setting up firewalls etc.
- **Server Provisioning** - As the name suggests it includes setting up a new server in the network. For example spinning up a new EC2 instance.
- **User Provisioning** - It manages identities and access to our network and individual components of a data centre. In AWS it could mean setting up IAM[8] roles and policies for databases or access to VPC.
- **Service Provisioning** - It includes the set up of a new service and managing the data associated with it. For example running a new container.

**Configuration.** On the other hand the development team usually expects the end machine to run some kind of operating system and to have other third party applications running on the given instance. Configuration can address these needs by creating a sequence of steps, that will configure new servers, start applications and overall prepare the environment. For example the most common use of configuration is when setting up NGINX.[9] On every machine you would want to install NGINX, configure what ports it handles and then you want to actually start the NGINX service. That is the job of configuration tools such as Ansible[10], again we will clarify this in a later section2.2.2

Another good reason to introduce automation to infrastructure is the ability to create identical or at least very similar environments.

**Environments.** When developing almost any software, as a developer you want to have a safe space to test your application. Of course you can do this on your local machine, but in the end that is not where the final application will run, so that raises a very common issue of developers saying "It works on my machine". We would like to create an environment for developers to play around with their code and also for it to be as similar as possible to the actual production environment, but it does not need as many resources or

[8]Identity and Access Management

[9]"NGINX is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more" [Tea16]

[10]`https://www.ansible.com/`

that it does not have to work without downtime.

We might also need a special environment for user testing by either testers or the actual client. Sometimes referred to as UA[11] or Testing environment.

We established that we need multiple very similar environments. Managing all of these environments manually is a very tedious task and when introducing changes to one environment, that will one day be projected to a different environment, it can lead to possible errors. Maybe we forget what changes were made to the environments, or we setup something slightly different and in the end our application does not start or can not communicate with other services. It would be much easier to have all of these changes to infrastructure managed by a set of configuration files and then have them automatically apply to a specific environment when we deploy a new version of our application that already knows how to work with these changes in infrastructure. That is where Infrastructure as a Code comes in and we will discuss it in more detail in section 2.2

### 2.1.4 Automation of Continuous Integration and Delivery

When developers code a new feature or fix a bug in their service, they usually version it with some sort of version control system, probably Git. Pushing new code to a Git repository means storing it somewhere and having backups or versions of that code, but that is all. In order to serve this new version of application we need to first execute a number of steps. These steps can include but are not limited to:

- **Checking the quality of code** - This step is not mandatory to the successful deployment of new application version, but is commonly a part of the process. It ensures that the newly added code follow some sort of style that the team agreed on and that it does not include any so called "code smells".
- **Automatic testing of the new application** - Either unit testing in a sandbox environment, or running end to end tests, or any other form of testing the application, that can be done by a computer.
- **Building application artifact** - The process of converting source code files into standalone software artifact or machine code, that can then be run on a server. Sometimes this can precede the testing step
- **Deploying built application** - After we have a fully tested and built application we need to copy the artifacts to our server. There are many strategies for deploying a new version of our application in order to reduce downtime including canary deployment, rolling updates, blue green deployment, or A/B testing strategy.

This is a very tedious and time consuming process which can be done by a computer. A decade ago these processes would be handled by a special set of

[11]User Acceptance

people, the IT Operations specialists, and it would take them a significant amount of time to get a new version running on a server. With automation we can achieve much faster and more frequent deploys of new versions. We also eliminate any possible errors that could occur if a human was doing this task. By triggering this task in a pipeline on a push to a Git repository we allow developers to in a way deploy their new code to a server alone and in the matter of minutes.

There is a whole new philosophy around deploying cloud based applications by keeping declarative descriptions of the infrastructure and application states in a Git repository, called GitOps, but again we will not discuss it in more detail as it is only an enclosing term for what we want to achieve with our system, but if the reader wishes to know more, we can recommend the book *GitOps - cloud- native continuous deployment* [BKH21].

We achieve this automatic process by introducing Continuous Integration and Continuous Deployment to our system

### 2.1.5 Environment Variables and Secrets

Microservices need to be able to run in different environments without modification to their code. We might have different databases for dev, stage and production environments, so we need to have a way of telling the service, how to connect to a specific database or any other third party service. We do this by externalising all configuration and reading the configuration at application startup from OS environment variables.

To visualise the concept of passing environment variables let's create an example. We could have a frontend and a backend application and we need to tell frontend how to access our backend. We can do this by passing a variable with the backend's endpoint. We also need to pass credentials for a PostgreSQL[12] database to the backend service. In Docker Compose[13] we can achieve such configuration with the following code 2.1:

**Source Code 2.1:** Docker Compose FE and BE configuration

```
version: '3'

services:
  frontend:
    build: ./frontend
    ports:
      - 3000
    depends_on:
      - backend
    environment:
      BACKEND_ENDPOINT: localhost:8080
```

[12]https://www.postgresql.org/docs/
[13]https://docs.docker.com/compose/

```
backend:
  build: ./backend
  expose:
      - 8080
  environment:
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
```

By exposing port `8080` from the backend container we can then access it on the local network as `localhost:8080`.

The advantage of this setup is that we do not need to build our frontend application every time we want it to communicate with a different backend. We can then only set the `BACKEND_ENDPOINT` variable to a different value when we deploy our application to a server for example `https://backend.example.com`.

**Secrets.** Secrets are an upgrade to environment variables. They work very similarly as environment variables and are usually passed to the application more or less the same way, but there is an added layer of security by encrypting these values

Secrets should be used when dealing with sensitive data for example credentials, API keys, and encryption keys. Another good practise is to retrieve secrets only once when starting our application to minimise possible leaks of these values.

As we speak about securing these values we must mention a common mistake with storing secrets and that is committing these values to a version control system. You should never commit secret values, or the encryption keys to retrieve them. Either projects get forked, or the communication while committing code could be interrupted, or you could unintentionally expose the `.git` folder. All of these possibilities are reasons to never commit secrets to repositories. Also we need to keep in mind, that if we once commit secrets to a repository and then delete them it does not mean they are gone forever. They are still kept in Git history and could be retrieved.

Because the stakes are high when storing environment variables and secrets, there are tools that help us manage them in a more effective and secure way such as Secrets Manager on AWS, or Vault by HashiCorp.

## 2.2 Infrastructure as a Code (IaC)

In the mid 2000s we saw an increase in complexity to applications. More hardware virtualisation and the rise of cloud providers allowed for cheaper infrastructure, but it also became more complicated to manage this infrastructure. It was a challenge to dynamically and frequently update your infrastructure and keep all environments as similar as possible.

The idea behind IaC is to write and execute code that defines, deploys, updates and destroys your infrastructure. We can split tasks of IaC tools into five different categories listed bellow:

- Ad-hoc scripts
- Configuration management tools
- Server templating tools
- Orchestration tools
- Provisioning tools

Let's look at them more closely.

### 2.2.1 Ad-Hoc Scripts

The simplest approach to automating almost anything is to write *ad-hoc scripts*. If possible you take any task you normally do manually and write a Bash, Ruby, or Python script to execute this task for you.

For example let's create a bash script to setup NGINX on a server:

**Source Code 2.2:** Bash NGINX Installation and Configuration

```bash
# Update dependencies
sudo apt-get update

# Install NGINX on our server
sudo apt-get install nginx

# Forward 'frontend.example.com' to 'localhost:3000'
echo "server {
        listen 80;
        server_name frontend.example.com;
        location / {
         proxy_pass http://localhost:3000;
       }
  }" > /etc/nginx/sites-available/default

# Start the NGINX service
sudo service nginx start
```

The benefits of this approach are:

- We can use general purpose programming languages to write scripts
- We have complete control over the steps that are executed and we can customise this process as much as we want.
- The task previously done manually is now scripted, and can be executed multiple times and should produce very similar outputs, unless the machine is in a different initial state then anticipated.

On the other hand even though we listed that we can write completely custom code for every task it is also a big downside as we *have to* write custom script for each task. The tools that are purposely built for IaC provide APIs for accomplishing much more complex tasks. If we had to maintain a big structure of bash scripts, the benefit of having IaC diminishes greatly.

To summarise: ad-hoc scripts are great for small tasks or tasks that are not executed frequently. But if we want to handle all of our infrastructure as a code, we need to use more sophisticated IaC tools.

### 2.2.2 Configuration Management Tools

Configuration management tools such as Ansible, Chef, Puppet are designed to install and manage software on existing machines. It sounds very similar to what ad-hoc scripts do, the difference is that these tools provide some convention advantages, but let's first create the same NGINX server using Ansible playbook:

**Source Code 2.3:** Ansible NGINX Playbook

```yaml
- name: Update dependencies
  apt:
  update_cache: yes

- name: Install NGINX
  apt:
  name: nginx

- name: Forward 'frontend.example.com' to 'localhost:3000'
  shell:  "echo `server {
    listen 80;
    server_name frontend.example.com;
    location / {
        proxy_pass http://localhost:3000;
       }
    }` > /etc/nginx/sites-available/default"

- name: Start NGINX
  service: name=nginx state=started enabled=yes
```

On its own we do not really see the benefits of using Ansible over ad-hoc scripts.[14]

By using such tool we gained these advantages:

- **Coding Conventions** - Ansible enforces consistent structure, file layout, and also is self documenting, which is one of the great benefits of IaC. So by enforcing some structure to the Ansible playbook we can on board

[14]Granted we did not introduce much customisation to the playbook as we wanted to keep the examples simple. A more clear and reusable way can be found in article [AN20]

new developers much more easily as they only need to know the structure of Ansible and not the structure, that someone created by writing a big pile of ad-hoc scripts.

- **Idempotence** - Writing an ad-hoc script, that works once is quite simple, but writing an ad-hoc script, that runs correctly even though you execute it multiple times is a much more difficult task. Even tasks as simple as creating a folder require you to first check if that folder already exists, the same goes with a new line of configuration, or when we started the NGINX service we should first check if the service is not already running.

  *Idempotent code* produces the same result without errors no matter how many times you run it. Most of Ansible's functions are idempotent by default and we no longer need to add many conditional statements to decide how our code should act in different situations. For example the above playbook 2.3 will only install NGINX if it is not already installed on the server.

- **Distribution** - By default Configuration management tools are designed specifically to be run on large numbers of remote machines.

  Again in Ansible we could accomplish this by first creating a file with the host IP addresses of the machines we want to execute the above playbook on.

  ```
  [webservers]
   3.80.11.11
   3.80.11.12
   3.80.11.13
   3.80.11.14
   3.80.11.15
  ```

  Next, you define the following Ansible playbook:

  ```
  - hosts: webservers
    roles:
     - webserver
  ```

  Executing this playbook will tell Ansible to configure all five servers in parallel.

### 2.2.3 Server Templating Tools

An alternative to configuration management tools are *server templating tools* such as Docker[15], Packer[16], and Vagrant[17]. These tools work more around the idea of packing a snapshot of the operating system, the necessary third

[15] https://docs.docker.com/
[16] https://www.packer.io/docs
[17] https://www.vagrantup.com/docs

party software, files, and all other relevant details in a so called *image* of a server. These images can be later installed on servers by a different IaC tool. This means we no longer need to setup multiple servers and then configure them by running the same code on each one.

For example the commonly used server templating tool is Docker. It creates an *image* of our system with all necessary third party services by executing a set of commands described in a Dockerfile. Then we can create multiple so called *containers* to run a copy of the image.

Let's create an example Dockerfile that will provide us with an image of alpine Linux running the NGINX service with a custom configuration as before.

First we create a *frontend.conf* file with the NGINX configuration:

```
server {
  listen 80;
  server_name frontend.example.com;
    location / {
    proxy_pass http://localhost:3000;
}
```

Then we define the actual Dockerfile 2.4:

**Source Code 2.4:** Dockerfile for NGINX Image

```
# Specifying a base image with only alpine linux installed
FROM alpine:latest

# Update the system packages
RUN apk update

# Install NGINX
RUN apk add nginx

# Copy the custom configuration for NGINX
COPY ./frontend.conf /etc/nginx/sites-available/default

# Specify the command to execute when running a container with
↪  this image
CMD ["service", "nginx", "start"]
```

We now need to build the snapshot into an image with this command:

```
docker build -t nginx-example
```

and then we can spin up a container with the `nginx-example` image with the following command:

```
docker run nginx-example
```

The obvious benefit of this approach is that we have everything necessary to run the application in one image and we only need Docker installed on a machine to run our application. Another benefit is that we can scale out our application by only spinning up new containers of this pre-built image.

Server templating became a major component in shifting to *immutable infrastructure*. It comes from declarative programming to create only immutable variables. If we need to update something in an image we need to create a new one. This eliminates unexpected errors that are hard to trace and enables us to debug and reason about our code.

### 2.2.4 Orchestration Tools

Now that we have images and containers created by server templating tools we need to actually manage them. To handle most use cases we will need to do some of the following steps:

- Deploy containers and make use of hardware efficiently
- Handle updates to existing containers e.g. rolling updates, blue-green deployment, canary deployment, and A/B testing strategy
- Monitor health of these containers and if necessary replace unhealthy instances by new ones
- Dynamically scale the number of running instances either vertically or horizontally (see 2.1.2)
- Distribute work load among containers (see 2.1.1)
- Allow these containers to talk to each other and talk to the outer world, or any third party services (see 2.1.1)

To achieve this we can use *orchestration tools* such as Kubernetes[18], Docker Swarm[19], Elastic Container Service[20], or Nomad[21]. One of the more popular tools is Kubernetes. Every major cloud provider has their managed way of running Kubernetes clusters (Elastic Kubernetes Service on AWS[22], Azure Kubernetes Service[23], Google Kubernetes Engine[24], DigitalOcean Kubernetes[25]). We would recommend starting with DigitalOcean Kubernetes because it is the cheapest out of the major providers and provides easy to use Dashboard for managing the cluster as well as a simple way of installing

[18]https://kubernetes.io/docs/home/
[19]https://docs.docker.com/engine/swarm/
[20]https://docs.aws.amazon.com/ecs/index.html
[21]https://www.nomadproject.io/docs
[22]https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html
[23]https://docs.microsoft.com/en-us/azure/aks/
[24]https://cloud.google.com/kubernetes-engine
[25]https://docs.digitalocean.com/products/kubernetes/

already prepared deployments such as Loki[26] or Cert-Manager[27].

Kubernetes provides us with a way of managing our containerised application through code. Although Kubernetes is very popular nowadays, from what we found out while developing our system, they are unnecessarily complex for smaller projects and Amazon Elastic Container Service provided us with seamless deployments as well as easy to use autoscaling. All of that only for the price of running server instances, while using Amazon Elastic Kubernetes Service for managed Kubernetes starts at $70 for the simplest cluster.

Once more we create an NGINX deployment with Kubernetes to introduce the technology:

**Source Code 2.5:** Kubernetes NGINX deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

This code snippet 2.5 was taken directly from Kubernetes documentation[28]. For further reading, please refer to that documentation.

### 2.2.5 Provisioning Tools

While all previously mentioned tools define the code to be run on a server, *provisioning tools* focus on creating servers themselves. You can also define how to provision databases, caches, load balancers, service mesh, queues, monitoring, firewalls, SSL certificates, DNS records, basically almost anything

[26] https://grafana.com/oss/loki/
[27] https://cert-manager.io/docs/
[28] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

you could think of when talking about infrastructure. Terraform 3.3.1, CloudFormation[29], Azure Resource Manager[30], OpenStack Heat[31] are all tools that try to simplify the process of provisioning infrastructure.

For example let's spin up a new EC2 instance running NGINX with Terraform 2.6.

**Source Code 2.6:** Terraform NGINX instance

```
resource "aws_instance" "app" {
  instance_type = "t2.micro"
  availability_zone = "eu-central-1a"
  ami = "ami-0c55b159cbfafe1f0"
  user_data = <<-EOF
    sudo service nginx start
  EOF
}
```

That's all there is to it. Sometimes it can be even faster to create resources with Terraform, then it is to create them manually through the AWS Console.

### 2.2.6 The Benefits of Infrastructure as a Code

Now that we saw all the possible types of tools used in IaC, we will provide some reasons for why we want to introduce IaC into the development process. Even though we already described some advantages of IaC in section 2.1.3 there are some other benefits to IaC worth mentioning:

- Clear and available infrastructure code for everyone on the team. There is no longer the need to have one system administrator, that handles all the magic behind closed doors.
- Safe and fast deployments are a direct product of automating the process. The safety part comes from eliminating human errors in the process of deployments.
- Self documenting. Again if we have infrastructure defined in code and following some conventions defined by the tool we are using, anyone can read it. All the knowledge about infrastructure is no longer locked in the head of one sysadmin.
- We can benefit from version control the same way we do while developing other software. This could mean having the ability to create merge requests and have changes recorded in history logs, so we can easily find out and roll back any changes that caused errors.

[29]https://aws.amazon.com/cloudformation/
[30]https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/overview
[31]https://wiki.openstack.org/wiki/Heat

- Having the state of our infrastructure defined in code gives us the ability to create automated tests for it, or run code analysis.
- Packaging infrastructure in reusable modules let's us create deployments for different environments without having to define new code for every environment from scratch.

# Chapter 3

# Technology Stack

## 3.1 Cloud Provider

The choice of cloud provider usually comes down to personal preference. There are three major cloud providers which provide basically the same services for very similar price. These are Amazon Web Services(AWS), Google Cloud Platform(GCP) and Microsoft Azure. We chose AWS because we have the most experience using it, and HashiCorp has an official resource provider with great documentation[1] for AWS, which makes it simpler to implement. Also there are many verified third party Terraform modules which helped us create a more reliable system.

We want to mention DigitalOcean as a good option when trying out cloud computing, as we found it on less expensive, especially when you want to get into Kubernetes. That said it feels much less polished and less reliable then any of the major providers. The lack of quality support from Terraform makes it an inferior choice when developing infrastructure as a code.

## 3.2 Orchestration

We wanted to avoid dealing with complex orchestration as it only really makes sense when creating large applications, that need a lot of custom behaviour. That was our biggest reason for not choosing Kubernetes. The initial setup is unnecessarily complicated to run a simple microservice application.

By choosing AWS we got the option to use Elastic Container Service, a simple orchestration tool that runs Docker containers without the need for manual intervention. That said, we vendor locked ourselves to AWS and because ECS only runs Docker containers we had to use Docker as a server templating tool. For some this could be a downside, but again we have experience with AWS. Docker is the most commonly used tool for server templating, and a lot of developers are familiar with it. With our system we are able to take any Docker image and run it on ECS and prepare it for public access, however we need the developers to specify how their code should be

[1]AWS provider `https://registry.terraform.io/providers/hashicorp/aws/latest/docs`

bundled, so they need to create the Dockerfile which assembles that image. We could say that being forced to use Docker is actually a benefit to us as the developers will be able to prepare their images for our system.

There are other orchestration tools such as Mesos[2], Docker Swarm[3] or Nomad[4], but all of these tools are again way too complicated to setup initially and are inadequate for our goal of simplicity.

## 3.3 Provisioning and Environments management

When deciding on a provisioning tool we can choose either from a variety of third party tools, or if we have previously chosen a specific cloud computing platform, then usually that platform has its own IaC provisioning tool. For AWS it is CloudFormation, for Azure it would be Azure Resource Manager and Google Cloud Platform have their Google Cloud Deployment Manager. These tools are well designed for their specific platform, but when it comes to migrating to a different cloud computing platform, we more or less have to implement the whole infrastructure from scratch. To avoid this problem we want to use a platform agnostic tool for example Terraform 3.3.1 or Pulumi 3.3.2.

However this is not the biggest issue we find with a tool like CloudFormation, the important part for us is, that it is too specific for the given provider, and you need to learn basically everything from scratch about how CloudFormation works when you try to read your infrastructure code. Platform agnostic tools provide a very similar interface and the same syntax for every provider.

To demonstrate this benefit, let's create an Elastic Compute Cloud instance and allow SSH access to it using CloudFormation 3.1 and compare it to Terraform code with the AWS provider plugin 3.2. Then are going to do the same for a virtual machine using Azure Resource Manager Template 3.3 and again compare it to Terraform code with Azure provider plugin 3.4

**Source Code 3.1:** CloudFormation EC2 setup

```
{
  "Resources": {
    "Ec2Instance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "SecurityGroups": [
          {
            "Ref": "InstanceSecurityGroup"
          },
          "MyExistingSecurityGroup"
```

[2]https://mesosphere.github.io/marathon/
[3]https://docs.docker.com/engine/swarm/
[4]https://www.nomadproject.io/docs

```
          ],
          "KeyName": "mykey",
          "ImageId": "ami-7a11e213"
        }
      },
      "InstanceSecurityGroup": {
        "Type": "AWS::EC2::SecurityGroup",
        "Properties": {
          "GroupDescription": "Enable SSH access via port 22",
          "SecurityGroupIngress": [
            {
              "IpProtocol": "tcp",
              "FromPort": 22,
              "ToPort": 22,
              "CidrIp": "0.0.0.0/0"
            }
          ]
        }
      }
    }
}
```

Example taken from official AWS documentation[5].

**Source Code 3.2:** Terraform EC2 setup

```
// This security group allows incoming traffic from every IP
↪  address on port 22 and outgoing traffic to every IP address
↪  and every port

resource "aws_security_group" "ingress-ssh-all" {
  name = "ingress-ssh-all"
  ingress {
    from_port = 22
    to_port = 22
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
   from_port = 0
   to_port = 0
   protocol = "-1"
   cidr_blocks = ["0.0.0.0/0"]
 }
```

[5] https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/gettingstarted.templatebasics.html

```
}

resource "aws_instance" "app" {
  instance_type = "t2.micro"
  availability_zone = "eu-central-1a"
  ami = "ami-7a11e213"
  // This line attaches the security group for ssh access
  security_groups =
  ↪  ["${aws_security_group.ingress-ssh-all.id}"]
}
```

**Source Code 3.3:** Azure Resource Manager VM setup

```
{
  "$schema": "https://schema.management.azure.com/schemas/
             2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "metadata": {
    "_generator": {
      "name": "bicep",
      "version": "0.4.613.9944",
      "templateHash": "7822315097766237434"
    }
  },
  "parameters": {
    "adminUsername": {
      "type": "string",
      "metadata": {
        "description": "Username for the Virtual Machine."
      }
    },
    "adminPassword": {
      "type": "secureString",
      "minLength": 12,
      "metadata": {
        "description": "Password for the Virtual Machine."
      }
    },
    "OSVersion": {
      "type": "string",
      "defaultValue": "2019-datacenter-gensecond",
      "allowedValues": [
        "2019-datacenter-gensecond",
        "2019-datacenter-core-gensecond",
        "2019-datacenter-core-smalldisk-gensecond",
        "2019-datacenter-core-with-containers-gensecond",
        "2019-datacenter-core-with-containers-smalldisk-g2",
        "2019-datacenter-smalldisk-gensecond",
        "2019-datacenter-with-containers-gensecond",
        "2019-datacenter-with-containers-smalldisk-g2",
        "2016-datacenter-gensecond"
      ],
      "metadata": {
        "description": "The Windows version for the VM.
        This will pick a fully patched Gen2 image of this given
↪   Windows version."
      }
    },
    "vmSize": {
      "type": "string",
      "defaultValue": "Standard_D2s_v3",
      "metadata": {
        "description": "Size of the virtual machine."
      }
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]",
      "metadata": {
        "description": "Location for all resources."
      }
    },
    "vmName": {
      "type": "string",
      "defaultValue": "simple-vm",
      "metadata": {
        "description": "Name of the virtual machine."
      }
    }
  },
  "variables": {},
  "resources": [
    {
      "type": "Microsoft.Compute/virtualMachines",
      "apiVersion": "2021-03-01",
      "name": "[parameters('vmName')]",
      "location": "[parameters('location')]",
      "properties": {
        "hardwareProfile": {
          "vmSize": "[parameters('vmSize')]"
        },
        "osProfile": {
          "computerName": "[parameters('vmName')]",
          "adminUsername": "[parameters('adminUsername')]",
          "adminPassword": "[parameters('adminPassword')]"
        },
        "storageProfile": {
          "imageReference": {
            "publisher": "MicrosoftWindowsServer",
            "offer": "WindowsServer",
            "sku": "[parameters('OSVersion')]",
            "version": "latest"
          },
          "osDisk": {
            "createOption": "FromImage",
            "managedDisk": {
              "storageAccountType": "StandardSSD_LRS"
            }
          },
          "dataDisks": [
            {
              "diskSizeGB": 1024,
              "lun": 0,
              "createOption": "Empty"
            }
          ]
        }
      }
    }
  ]
}
```

Example taken directly from Microsoft Azure Quick-start guide, but without network resources [6]

**Source Code 3.4:** Terraform VM setup

```
variable "prefix" {
  default = "tfvmex"
}

resource "azurerm_resource_group" "main" {
  name     = "${var.prefix}-resources"
  location = "West Europe"
}

resource "azurerm_virtual_machine" "main" {
  name                  = "${var.prefix}-vm"
```

[6] https://docs.microsoft.com/en-us/azure/virtual-machines/windows/quick-create-template

```
  location              = azurerm_resource_group.main.location
  resource_group_name   = azurerm_resource_group.main.name
  vm_size               = "Standard_DS1_v2"
  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }
  storage_os_disk {
    name              = "myosdisk1"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Standard_LRS"
  }
  os_profile {
    computer_name  = "hostname"
    admin_username = "testadmin"
    admin_password = "Password1234!"
  }
  os_profile_linux_config {
    disable_password_authentication = false
  }
}
```

Example taken from Terraform documentation for Azure provider, but again without network resources[7]

From these examples we can clearly see that Terraform enforces the use of unified syntax and in our opinion the code is more clean and readable. Even though we created server instances for different platforms, in Terraform we use the same keywords for example *variable* and *resource*.

There are not many *purely* provisioning tools other then Terraform and Pulumi. You can use most of the Configuration Management Tools such as Ansible for provisioning, but it is not optimal to do so. Let's look at Terraform and Pulumi in more detail

### 3.3.1 Terraform

Terraform is an open source Infrastructure as a Code tool created by HashiCorp. It was written in the Go programming language which compiles down to into a single binary called `terraform`.

Under the hood the `terraform` binary makes API calls to a *provider* such as AWS, Azure, Google Cloud, OpenStack and more. This means that

[7] `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/virtual_machine`

Terraform gets to leverage from the providers existing infrastructure for their API servers and use authentication mechanisms we are already using with this provider e.g. API keys for our account at AWS.

Our job is now to declare how should Terraform execute these API calls. We use the word *declare* as Terraform encourages a more *declarative* style in which we write code that specifies the desired state of our infrastructure and Terraform it self is responsible for figuring out how to achieve that state. On the other hand Chef and Ansible encourage a *procedural* style in which we write code that specifies, step by step, how to achieve the desired state.

Let's demonstrate this difference from which we will clearly see, why declarative approach is a benefit. If we wanted to deploy 5 servers (EC2 instances on AWS) we can do this with Ansible using *procedural* approach as follows:

```
- ec2:
  count: 5
  image: ami-7a11e213
  instance_type: t2.micro
```

And here is the corresponding Terraform configuration that does the same thing using *declarative* approach:

```
resource "aws_instance" "example" {
  count         = 5
  ami           = "ami-7a11e213"
  instance_type = "t2.micro"
}
```

At first glance, these two approaches look similar, and when we execute them, they will initially produce the same results. The interesting thing is what happens when we try to update this configuration. For example if we wanted to increase the number of servers from 5 to 15. The procedural code is no longer viable as it says *create 5 t2.micro instances with the specified ami.* If we only changed the *count* from 5 to 15, Ansible would again create 15 t2.micro instances. But because this is a procedure, Ansible does not care about the 5 previously created server, which results in us having 20 servers instead of 15. So to reach the desired goal of having 15 servers we would have to write this code:

```
- ec2:
  count: 10
  image: ami-7a11e213
  instance_type: t2.micro
```

With declarative code, all we do is declare the end state that we want, which is to *have 15 t2.micro instances*, and we let Terraform figure out how to reach this state. Terraform is aware of the 5 previously created servers and will deploy only 10 additional servers. We do this by editing the same Terraform configuration:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-7a11e213"
  instance_type = "t2.micro"
}
```

With declarative approach we gained two major advantages:

- **Fully captured end state of our infrastructure** - Reading through the Ansible templates is not enough to know what is deployed in the end. We also need to know in which order were these templates applied. With our example it would not have mattered in which order we execute them, but we can definitely imagine a more complicated situation in which the order would result in different infrastructure. With declarative approach, the code always represents the latest state of the infrastructure.
- **Re-usability** - In procedural code we had to manually take into account the current state of the infrastructure. Because the state is constantly changing the procedural code quickly becomes nonfunctional as it was designed to modify a state of the infrastructure which no longer exists. With declarative approach, we let Terraform account for the current state, and only describe the next end state.

To summarise, by using Terraform we gain:

- platform agnostic syntax for our infrastructure code
- self-documenting infrastructure
- reusable infrastructure code

There are two key concepts when working with Terraform.

**Reusable modules.** The concept of reusable modules falls back to the idea of loose coupling. In Terraform a module is a set of multiple resources that are used together and logically relate to each other. For example when creating a Virtual Private Cloud we want to also create subnets and security groups. Having all of this as one reusable module gives us the option of creating multiple VPCs by only passing different variables to this module.

We do not want to go more in depth about this topic as this thesis is not a guide on how to write Terraform infrastructure, so we will refer the reader to the official Terraform documentation[8] for further reading.

**State management.** The second import concept in Terraform is how to keep track of the current state of infrastructure. The Terraform state keeps track of the latest state of our infrastructure as well as creates a link between our infrastructure code and the actual corresponding resources. For example this configuration:

[8]https://www.terraform.io/language/modules/syntax

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

will produce this *terraform.tfstate* file:

```
{
  "version": 4,
  "terraform_version": "0.12.0",
  "serial": 1,
  "lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
  "outputs": {},
  "resources": [{
    "mode": "managed",
    "type": "aws_instance",
    "name": "example",
    "provider": "provider.aws",
    "instances": [{
      "schema_version": 1,
      "attributes": {
        "ami": "ami-0c55b159cbfafe1f0",
        "availability_zone": "us-east-2c",
        "id": "i-00d689a0acc43af0f",
        "instance_state": "running",
        "instance_type": "t2.micro",
        "(...)": "(truncated)"
      }
    }]
  }]
}
```

This maps the `aws_instance` with name `example` to the corresponding AWS EC2 instance with id `i-00d689a0acc43af0f`.

When dealing with `terraform.tfstate` we need to be careful as it stores sensitive data about our infrastructure, so we never want to commit it to Git. However if we can not commit Terraform state to version control we need to find a secure way of sharing this state with other members of the development team. This is handled by remote Terraform state and HashiCorp provides a secure place for storing your state called Terraform Cloud, but other options such as keeping state on Amazon Simple Storage Service(S3) or Azure Blob Storage are possible and because we are already using AWS we chose the option of storing Terraform state on S3.

Both of these concepts are not well managed by Terraform on its own, so we will use Terragrunt 3.3.1 to handle them more effectively.

### Terragrunt

Terragrunt is an open source wrapper for Terraform which provides tools for keeping our infrastructure code DRY(Don't Repeat Yourself). It can leverage Terraform's reusable modules as well as manage the remote state.

Terragrunt is a thin wrapper for Terraform, which means after we install it, we can run all the standard `terraform` commands using the `terragrunt` binary:

```
$ terragrunt init
$ terragrunt validate
$ terragrunt plan
$ terragrunt apply
```

The only difference is that Terragrunt also uses any configuration specified in a *terragrunt.hcl* file. This file provides us with some extra behaviour. The basic idea is that we define infrastructure with Terraform code only once in reusable modules, and then use *terragrunt.hcl* files to configure these modules for different environments, thus enforcing the DRY principle onto our infrastructure code.

Let's look at an example. We want to setup an EC2 instance connected to a PostgreSQL database managed by AWS Relational Database Service(RDS):

**Source Code 3.5:** Terraform code for EC2

```
variable "count" {
  default     = 1
  description = "Number of ec2 instances to setup"
  type        = number
}

variable "instance_type" {
  default     = "t3.micro"
  description = "The instance type for ec2 instance"
  type        = string
}


resource "aws_instance" "example" {
  count         = var.count
  ami           = "ami-7a11e213"
  instance_type = var.instance_type
}
```

**Source Code 3.6:** Terraform code for RDS

```
variable "allocated_storage" {
  default     = 10
  description = "Allocated storage for RDS instance in GBs"
  type        = number
}
variable "instance_class" {
  default     = "db.t3.micro"
  description = "The instance class for db instance"
  type        = string
}

resource "aws_db_instance" "db" {
  allocated_storage = var.allocated_storage
  engine            = "postgres"
  engine_version    = "11.10"
  instance_class    = var.instance_class
}
```

We can imagine variables the same way as in normal software development, only writing the definition is a bit more complex. However introducing variables into our code gives us the option to configure resources for each environment differently.

Now we can use Terragrunt to setup *dev* environment with 1 EC2 instance using the *t3.micro* instance type and allocate only 5 GB of storage to our database and the *production* environment with 5 EC2 instances using the *t3.medium* instance type and allocate 50 GB of storage for the database. Both of theses environments will use the same Terraform modules defined in 3.5 and 3.6:

**Source Code 3.7:** Terragrunt EC2 configuration for Dev Environment

```
inputs = {
  instance_type = "t3.micro"
  count = 1
}
```

**Source Code 3.8:** Terragrunt RDS configuration for Dev Environment

```
inputs = {
  instance_class = "db.t3.micro"
  allocated_storage = 5
}
```

And the production environment configuration:

**Source Code 3.9:** Terragrunt EC2 configuration for Production Environment

```
inputs = {
  instance_type = "t3.medium"
  count = 5
}
```

**Source Code 3.10:** Terragrunt RDS configuration for Production Environment

```
inputs = {
  instance_class = "db.t3.micro"
  allocated_storage = 50
}
```

Splitting the code into two modules gives us more flexibility when setting up the infrastructure, but also introduces an issue with dependencies. In our example we would like to always first create the RDS instance before creating any EC2 instances. We can achieve this with Terragrunt with the `dependency` keyword as follows:

**Source Code 3.11:** Terragrunt EC2 configuration with RDS Dependency

```
dependency "rds" {
  config_path = "/data-storage/rds"
}

inputs = {
  instance_type = "t3.medium"
  count = 5
}
```

Now we have the sequence in which we want to apply the code. Terragrunt lets us execute the `terraform apply` command on multiple modules at the same time by providing the `terragrunt run-all apply` command. This will go through all the modules with *terragrunt.hcl* file inside and create the desired sequence before applying the configuration.

Another advantage of using Terragrunt is the DRY management of remote state and Terraform provider. We are able to create a common *terragrunt.hcl* file 3.12 in the *root* directory, which defines these settings:

**Source Code 3.12:** Root Terragrunt configuration file

```
generate "provider" {
  path = "provider.tf"
  if_exists = "overwrite_terragrunt"
  contents  = <<EOF
    provider "aws" {
```

```
    region = "eu-central-1"
  }
  EOF
}

remote_state {
  backend  = "s3"
  generate = {
    path      = "backend.tf"
    if_exists = "overwrite_terragrunt"
  }

  config = {
    bucket         = "terraform-state"
    key            =
     ↪  "${path_relative_to_include()}/terraform.tfstate"
    region         = "eu-central-1"
    encrypt        = true
    dynamodb_table = "terragrunt-lock-table"
  }
}
```

We will not explain in detail how this code snippet works as it is part of the official documentation[9].

We can see that Terragrunt uses the same language as Terraform which is the HashiCorp Language(HCL)[10]. It keeps the syntax for provisioning similar to pure Terraform code.

To summarise, these are the benefits of using Terragrunt:

- simple configuration of different environments with DRY modules
- handling of module dependencies
- DRY management of provider and remote state
- similar syntax to using pure Terraform code
- one command to deploy and destroy the entire infrastructure

### 3.3.2 Pulumi

Pulumi[11] is an open source[12] Software Development Kit(SDK) for managing infrastructure as a code. It let's you define infrastructure as a code by using familiar programming languages. This brings the concept of defining infrastructure through code closer to the end developer.

[9] https://terragrunt.gruntwork.io/docs/features/keep-your-remote-state-configuration-dry/
[10] https://github.com/hashicorp/hcl
[11] https://www.pulumi.com/docs/
[12] https://github.com/pulumi/pulumi

Pulumi works with traditional infrastructure such as VMs, network, databases, server instances, but it can also provide solutions for containerised applications, Kubernetes orchestration and many more.

We can use different languages to work with Pulumi such as TypeScript, JavaScript, Python, Go or C#. Let's once again create an EC2 instance with SSH access, this time using Pulumi 3.13 with JavaScript;

**Source Code 3.13:** Pulumi NGINX setup

```
import * as aws from "@pulumi/aws";
import * as pulumi from "@pulumi/pulumi";

const instanceType = "t2.micro";
const ami = aws.getAmiOutput({
    filters: [{
        name: "name",
        values: ["amzn-ami-hvm-*"],
    }],
    owners: ["137112412989"], // This owner ID is Amazon
    mostRecent: true,
});

const group = new aws.ec2.SecurityGroup("security-group", {
    ingress: [
        { protocol: "tcp", fromPort: 22, toPort: 22,
         ↪  cidrBlocks: ["0.0.0.0/0"] },
    ],
});

const server = new aws.ec2.Instance("webserver", {
    instanceType,
    vpcSecurityGroupIds: [ group.id ], // reference the
     ↪  security group resource above
    ami: ami.id,
});

// Outputs for the newly created server
export const publicIp = server.publicIp;
export const publicHostName = server.publicDns;
```

We can see, that using JavaScript to code our infrastructure looks the same as coding any other application.

Looking back it would be beneficial to use Pulumi instead of Terraform for our system, because the other developers would be able to read and use our infrastructure code even more easily. Unfortunately at that time Pulumi was only in version 2 and we had some issues for example with creating Task

Definitions for Elastic Container Service. Also we had more experience with Terraform so it felt like the better choice. Even with some previous issues, we feel like Pulumi is a great SDK, and we would probably use it in a future project over the Terraform, Terragrunt combination.

## 3.4 Continuous Integration and Delivery

The choice of a tool that provides us with Continuous Integration and Continuous Delivery(CI/CD) was quite simple. There were two criteria that affected our choice: simplicity of the tool and our experience using it.

We will divide CI/CD tools into two categories:

- **Third party self-hosted tools** - Tools that are separate from our Git environment and usually need to be self-hosted. In this category we have very popular tools such as Jenkins[13], CircleCI[14], Agola[15], AWS CodeBuild[16] and many more[17].
- **Tools from Git providers** - In the second category we put tools that are implemented by the chosen Git provider and directly integrated into their service. These tools can be self-hosted, however usually the providers give us the option to use shared resources to run CI/CD pipelines. In this category there are three major Git providers and their tools. GitHub has their GitHub Actions[18] tool, Bitbucket let's their users use Bitbucket Pipelines[19], and GitLab implemented a tool called GitLab CI/CD[20]

Now we said the choice was simple, because all of the self-hosted third party CI/CD tools are more complicated to setup then the ones implemented by Git providers. That leaves us with Bitbucket, GitHub and GitLab. All of these providers and their tools can be used for our purposes, so the choice comes down to our experience and also our choice of Git provider. We went with *GitLab CI/CD* as we have been using it for the past four years on personal projects and are quite content with its features and syntax.

[13] `https://www.jenkins.io/doc/`
[14] `https://circleci.com/docs/`
[15] `https://agola.io/doc/`
[16] `https://docs.aws.amazon.com/codebuild/index.html`
[17] A detailed list of CI/CD tools can be found in the *awesome-ci* GitHub repository
[18] `https://docs.github.com/en/actions`
[19] `https://bitbucket.org/product/features/pipelines`
[20] `https://docs.gitlab.com/ee/ci/`

# Chapter 4

# System Design

Figure 4.1 illustrates most of the common requirements for infrastructure. A cloud web application most likely consists of a frontend or Web UI[1] component 4.1, followed by a backend application which consists of one or more microservices 4.3. These services may need other resources usually data storage 4.4 such as relational database or cache storage. This backend application should be only accessible to authorised users 4.2. A very common requirement is to store logs from all of the resources in the system as well as to monitor them in real time 4.5. In section 2.1.5 we talked about external configuration of services through environment variables, so we provide the development team with the ability to edit these values 4.6. Last but not least a major part of every software development is the delivery of new code updates 4.7.

[1]User Interface

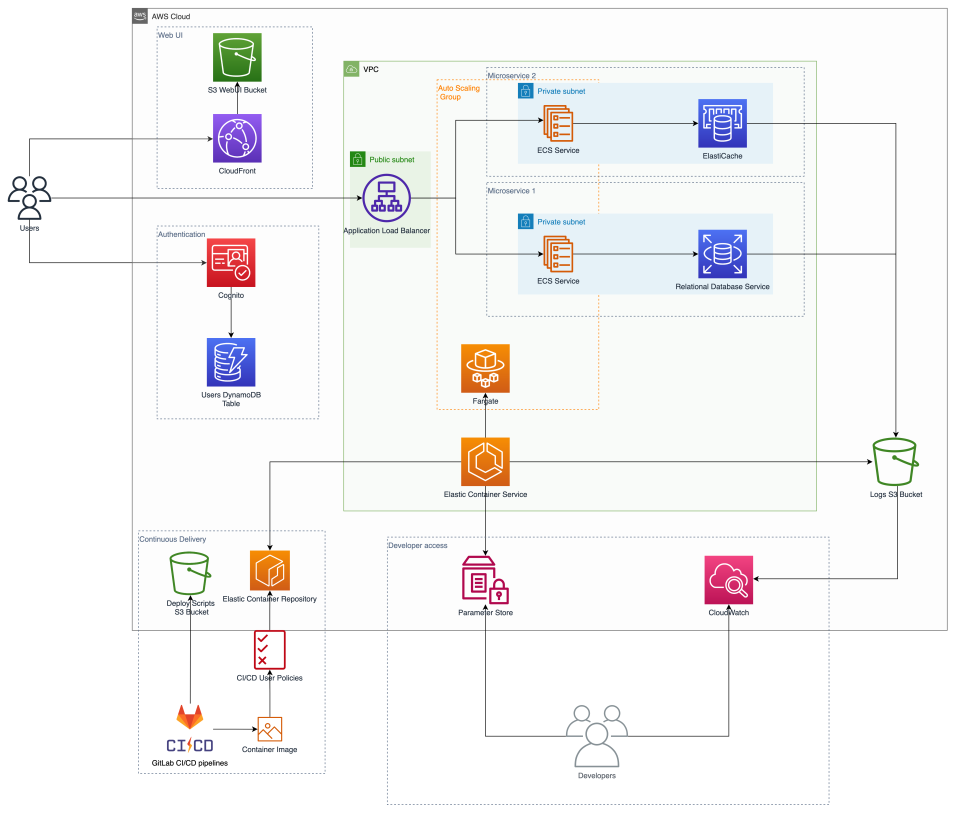


**Figure 4.1:** Infrastructure Diagram

The above diagram 4.1 shows all the described components, however it is quite complex. Let's describe the design in smaller parts:

## 4.1 Web UI

The Web UI component of our system is quite simple and also optional. Nowadays it is very common to build frontend applications into a set of static files. These files usually include all necessary JavaScript, HTML and CSS code to run the application. We can then store these files for example on S3[2]. Then we let users access these static files through a Content Delivery Network(CDN) such as CloudFront[3]. Having a CDN provides us with out of the box caching of these files, or other assets.

[2]Simple Storage Service
[3]`https://docs.aws.amazon.com/cloudfront/index.html`

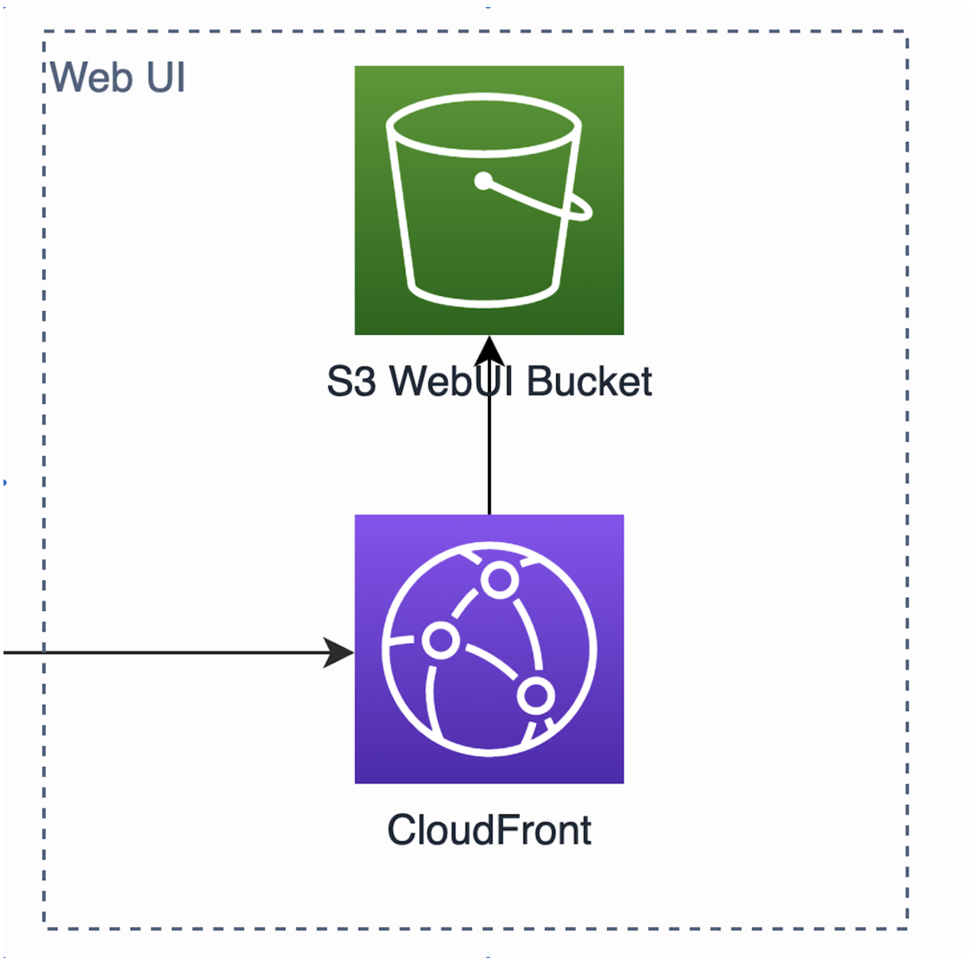


**Figure 4.2:** Web UI component

Figure 4.2 visualises this setup.

Another very common way of serving frontend applications is by running a small web server, similar to a backend service, which accepts HTTP requests for the files. In our system we can solve this by adding a microservice, the same way as any other, see 4.3.

## 4.2 Authorisation

In section 2.1.1 we talked about the importance of securing our application. One of the components was allowing only authorised users to use our application. We can manage these users with AWS Cognito[4]. It is possible to let Cognito store the users, but usually we want to add more information to the user entity or use it elsewhere in our application. That is why we add a NoSQL[5] database such as DynamoDB[6], to store user entities. This setup is

[4]`https://docs.aws.amazon.com/cognito/index.html`
[5]`https://cs.wikipedia.org/wiki/NoSQL`
[6]`https://docs.aws.amazon.com/dynamodb/index.html`

shown in figure 4.3

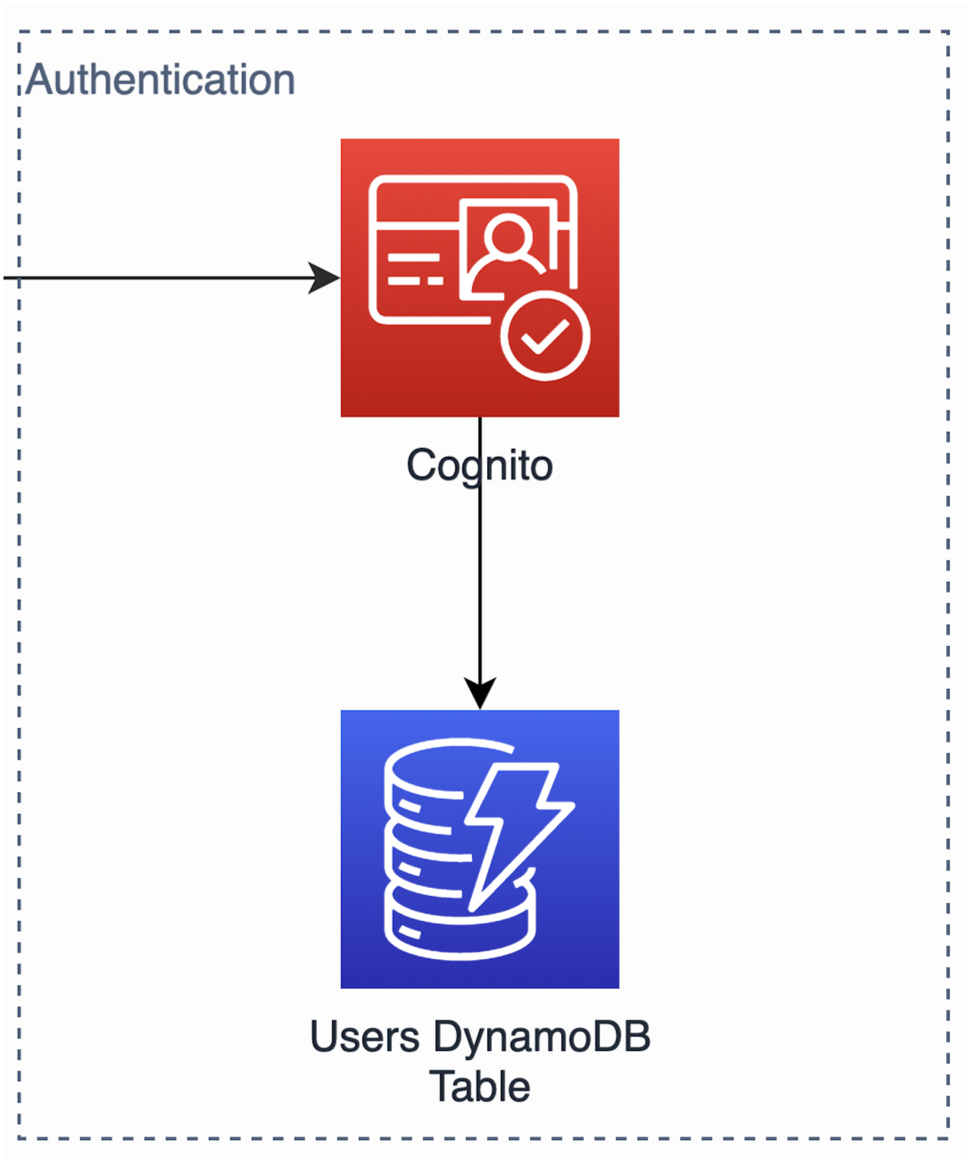


**Figure 4.3:** Authorisation component

Again some projects already use their own methods of authorising and authenticating users with a special microservice to handle these operations. That setup is again not a problem for our system and can be achieved by running a microservice setup 4.3.

## 4.3 Microservices

The core of our system lies in the definition of individual microservices. In section 3.2 we discussed different orchestration tools and opted for Elastic

Container Service(ECS). It allows us to create *task definitions*[7] which describe what Docker image to use, and how much resources we want allocated to our service. However we need to keep in mind, it is only a definition of how the end service should look like and we need to specify where to get these resources. AWS provides us with two main options for server instances: regular EC2 instances and Fargate[8]. When choosing the route with EC2 instances you need to manage them manually. This means you need another configuration for scaling the instances both horizontally or vertically. We chose the second option with using Fargate. It is a fairly new concept in AWS and it provides resources on demand, meaning we only need to specify, how much resources we want for our service, and let Fargate provide them. ECS can then request these resources when it creates new services, figure 4.4.

[7] `https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html`

[8] `https://aws.amazon.com/fargate/`

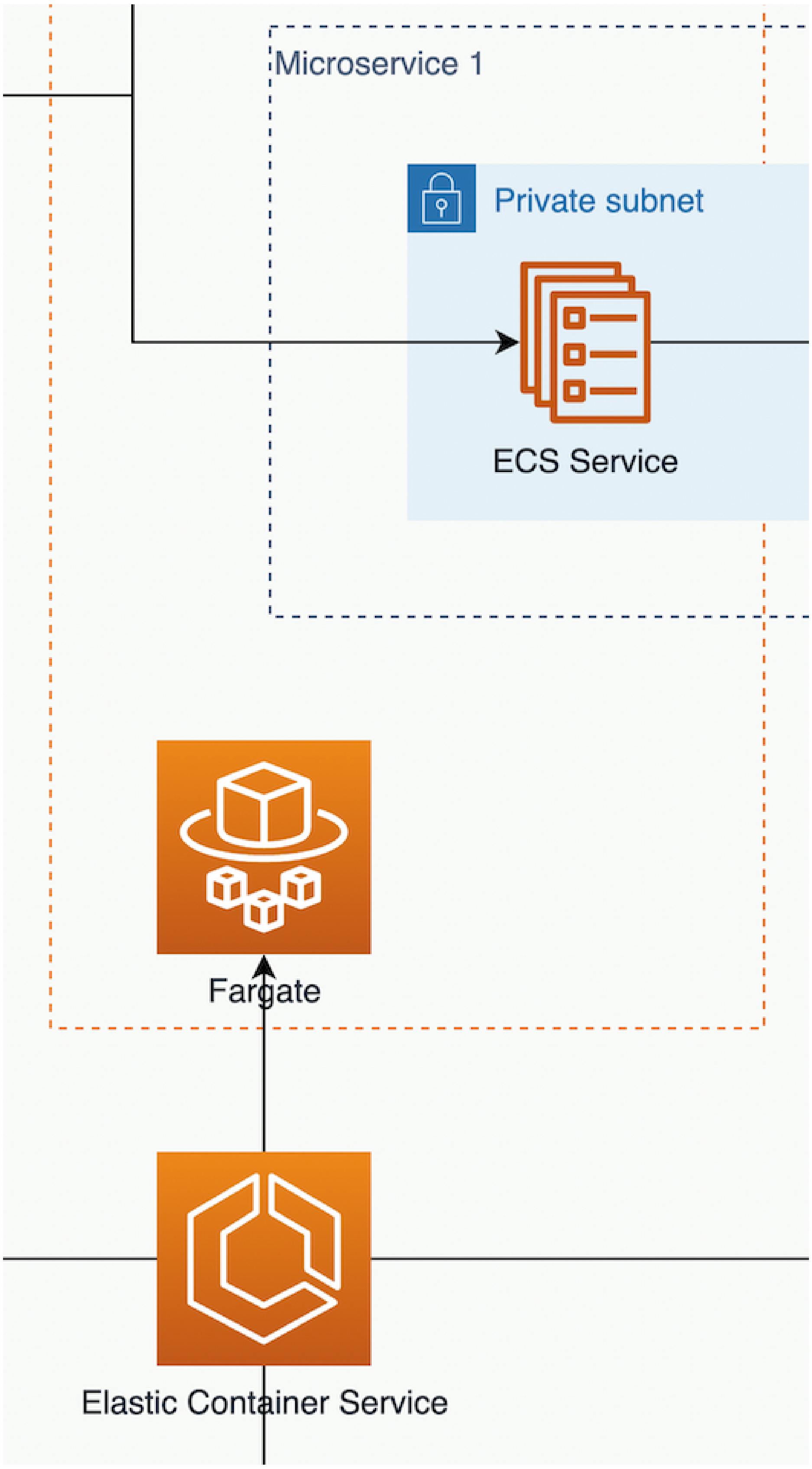


**Figure 4.4:** Elastic Container Service with Fargate

In the second figure 4.5 we show an example of how a setup of two microservices could look like. Both of these services are managed by ECS and have their private subnet. The first service also communicates with a Relational Database on its network, while the second one uses an in memory ElastiCache such as Redis[9]. This is a basic setup and we designed the system to be able to create new microservices on demand, by creating new Terragrunt configuration files, which use the same base ECS module 5.1.2. The microservices are not limited to one private subnets, they can be in multiple ones allowing them to either share database, or communicate with each other, if they lie in the same subnet.

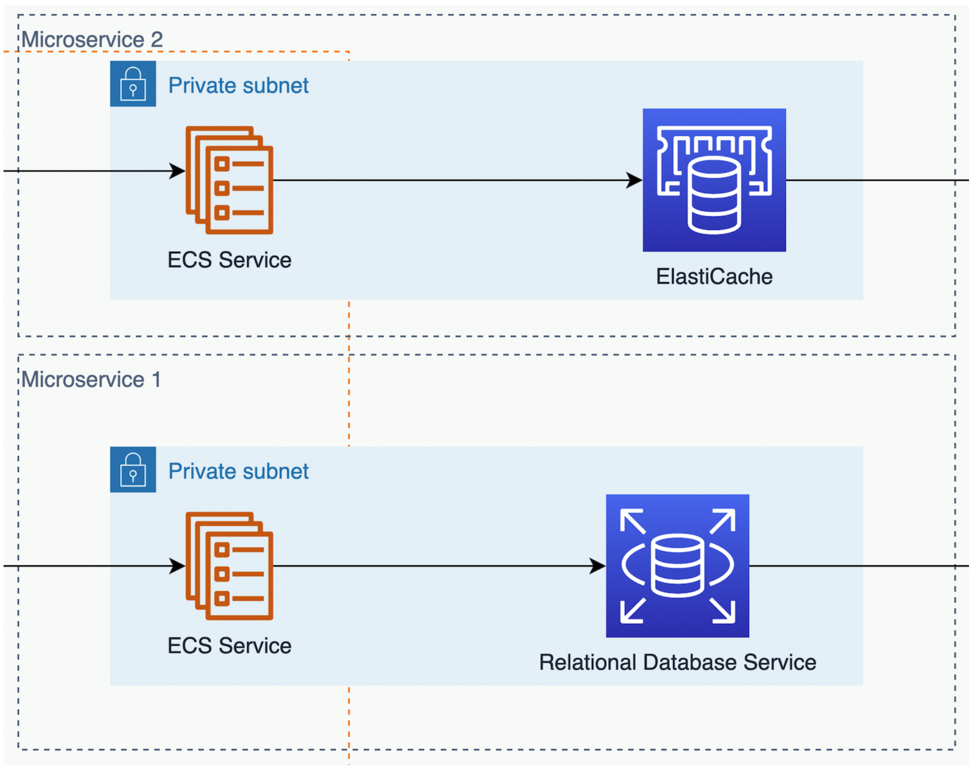


**Figure 4.5:** Microservices component

## 4.4 Data storage

Currently the system is designed to support two types of data storage 4.5: Relational Databases, which by default run on the PostgreSQL engine, and ElastiCache clusters, by default running Redis. Individual microservices can connect to this storage instances by exposing endpoints and credentials with environment variables 4.6. We expect users of our system to use relational databases for persistent data storage, and ElastiCache for caching and storing temporary data such as user sessions [Ora].

[9] `https://redis.io/`

## 4.5 Logging and Monitoring

All components in our system, that we described up to this point log either directly into CloudWatch, or send logs to a S3 bucket, from which we can then import logs and metrics into CloudWatch when needed, figure 4.6.

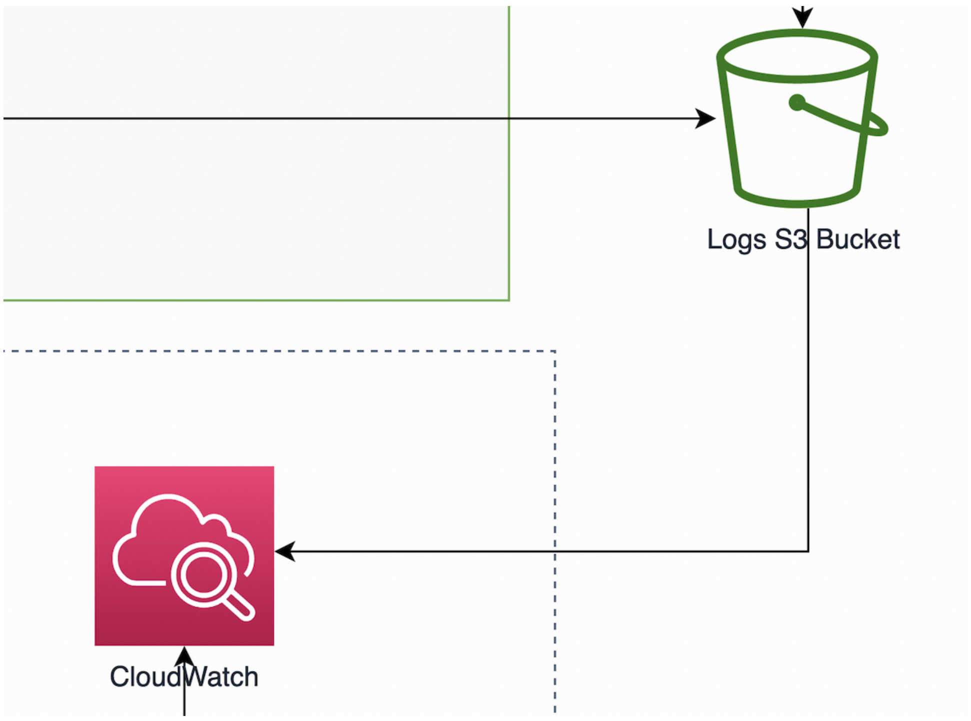
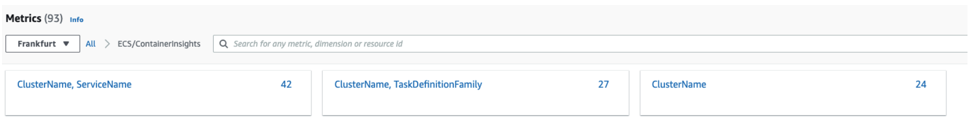


**Figure 4.6:** Logging and Monitoring

Every project has their specific needs for what it needs to monitor, so we do not create any metrics or graphs with our system as every metric in CloudWatch costs around \$0.3, which might seem like a small amount, but it quickly adds up. To give the reader a sense of how quickly the number of metrics can grow, in figure 4.7 we show a real world metrics count in a setup of two microservices in ECS and two environments.



| | |
|---|---|
| ClusterName, ServiceName | 42 |
| ClusterName, TaskDefinitionFamily | 27 |
| ClusterName | 24 |

**Figure 4.7:** Metrics count

93 metrics at \$0.3 each equals to \$27.9 a month just for metrics. To put that in perspective, an ECS service with 2GB of RAM and 1 vCPU running on Fargate for 28 days costs roughly the same amount. That is why we let users create their metrics on their own. However it could be a nice future

feature to analyse the most common and necessary metrics, so we could create essential metrics for every project.

## 4.6 Environment variables storage

In section 2.1.5 we discussed the need to configure services for different environment without adjusting their code. We use AWS Parameter Store[10] to store and inject environment variables and secrets into microservices. We will show how we setup individual parameters and link them to corresponding services dynamically in section 5.1.2

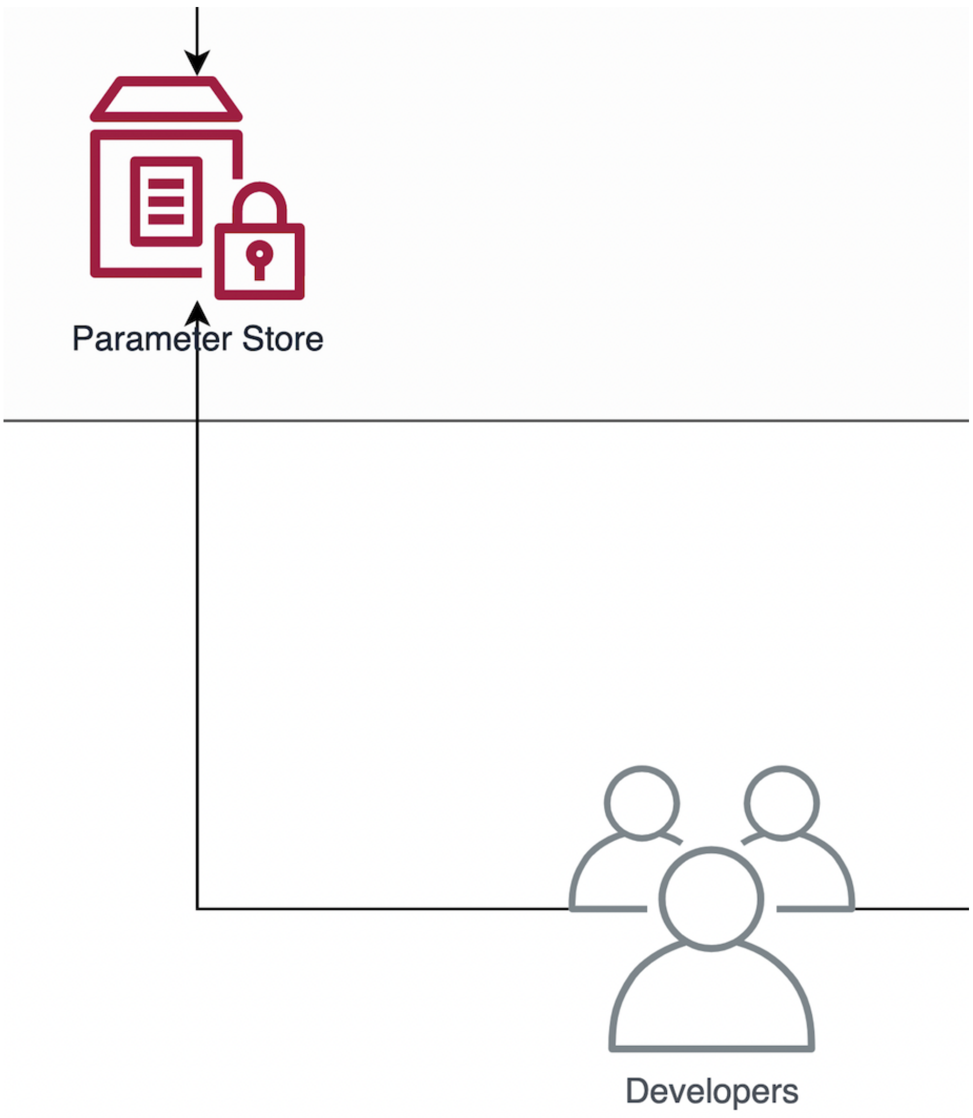


**Figure 4.8:** Parameter Store Component

[10]`https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-parameter-store.html`

The process of adding environment variable looks as follows:

- (Manual) create new parameter in AWS Parameter Store with Terragrunt
- (Automated) update task definition, to link this new parameter to an ECS service
- (Manual) an user responsible for managing environments sets the value for this new parameter
- (Automated) we update ECS to follow this new task definition

Unfortunately this process still involves manual adjustments to the infrastructure code, especially when creating new parameters. In the future, we would like to propose a common principle of defining environment variables in version control without the actual values, so we could dynamically create parameters in Parameter Store.

## 4.7 Continuous Integration and Delivery

The CI/CD part of our system begins with developers pushing their code to version control with a tag on commit and ends with an updated version of the application running on an environment. We can summarise the flow of this process with the following steps:

- Push code to version control with a tagged commit.
- Check quality of code and run lint jobs.
- Create a temporary Docker image with all development dependencies.
- Use this Docker image to run unit tests and if available run e2e tests.
- Use the same image to build the application and create a distribution folder with production files.
- Create a new Docker image with only production dependencies and copy the distribution file from previous step to this image.
- Tag the image with the commits tag (usually version tag e.g. `v1.0.0`)
- Upload the final image to Elastic Container Registry
- Update task definition of the ECS service to use this new image when creating new services.
- Redeploy the ECS service definition with new task definition to a specific environment. We leave this process as a manual trigger from the Gitlab Pipelines tab, so developers can deploy their code when they want to.
- ECS rolls the new update and when it finishes, it destroys the outdated service.

The steps above 4.7, are mostly just sequences of shell commands to be executed. We needed to design the system and the scripts to be reusable in all other project repositories so we upload these scripts to an S3 bucket, and we can download and use them in individual project repositories.

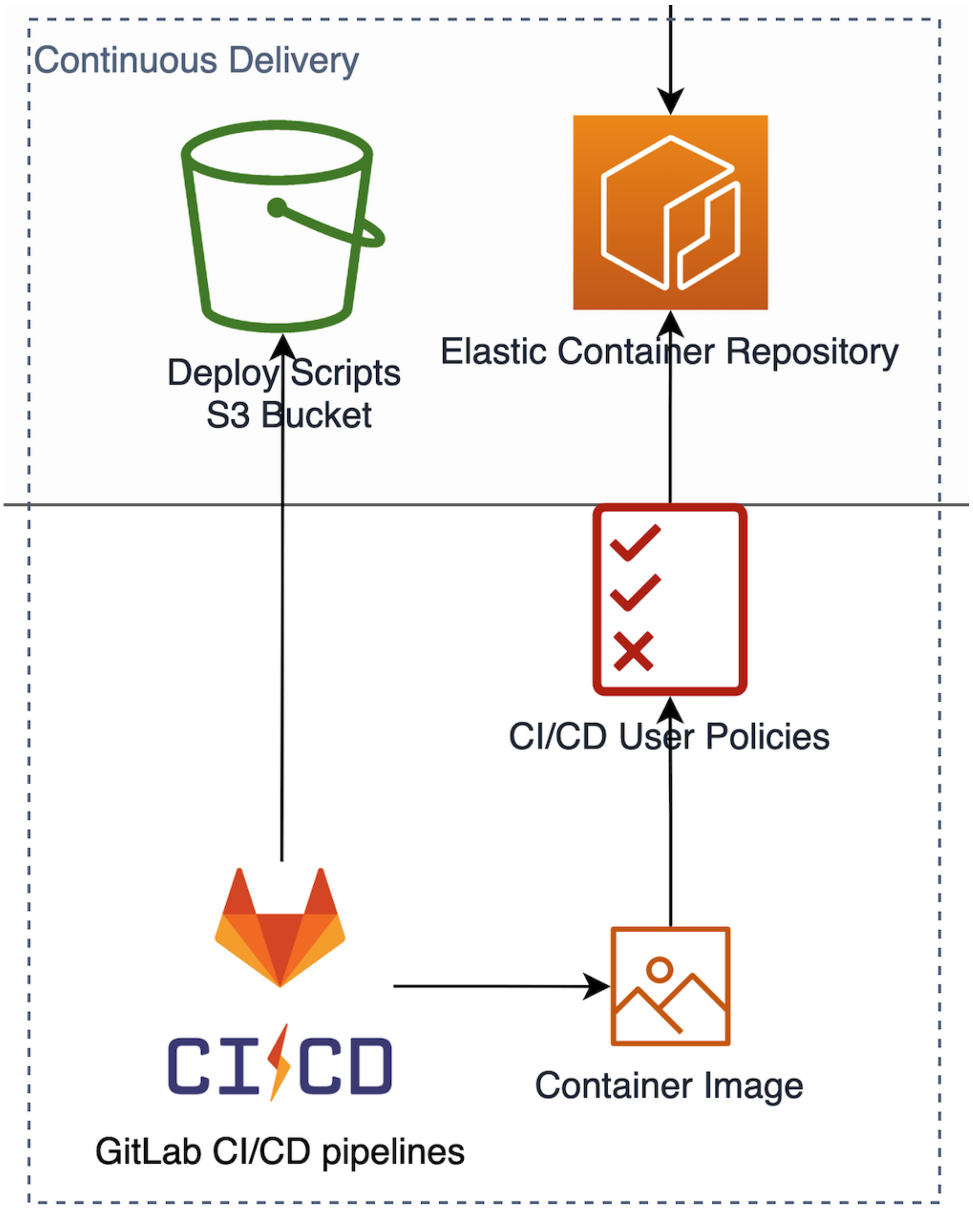


**Figure 4.9:** Continuous Delivery

Running Continuous Integration and Delivery also needs its own infrastructure. For now we can use the GitLab shared runners to run our CI/CD pipelines[11]. A good next step for our system would be to create our own EC2 instance and setup a managed GitLab Runner specific to our project.

In figure 4.9 we show the setup outside of Gitlab CI/CD. It consists of a S3 bucket, to store and transfer our bash scripts. From the CI/CD pipeline we create container images, those get stored in Elastic Container Registry.

An important part of the Continuous Delivery component are the CI/CD User Policies. We need to provide GitLab CI/CD with programmatic access to our AWS account, however we need to restrict its access to only a small set of operations such as download and upload files to S3, or manage ECS services and task definitions.

[11]At the time of writing this thesis, GitLab is updating their policy regarding free use of shared runners, because they had problems with people using their shared runners to mine cryptocurrencies. As of June 2022, GitLab will provide 400 pipeline minutes only to public projects and projects that enrol in their Public Open Source Program `https://about.gitlab.com/blog/2022/02/04/ultimate-perks-for-open-source-projects/`

# Chapter 5

# Implementation Process

## 5.1 Infrastructure Provisioning

The implementation of provisioning infrastructure consists of Terraform and Terragrunt configuration files. All of the files are located in the attachment `infrastructure/provisioning/`.

### 5.1.1 Global Modules

The global modules folder handles provisioning of resources shared among environments. When creating new versions of the application i.e. creating new container images, we upload images to ECR, that is shared among all environments as they all need to pull the same image.

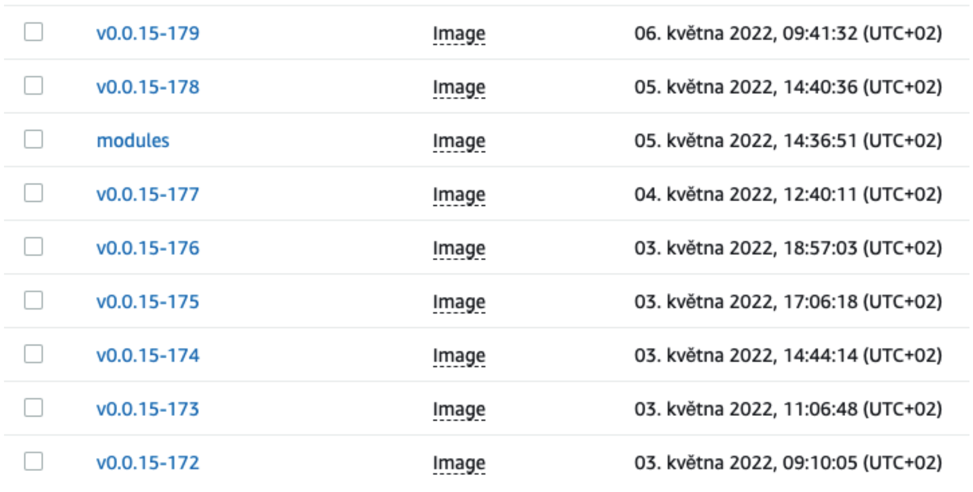
**Elastic Container Registry.** Setting up ECR is done very simply by telling Terraform to create an repository for our microservice.

```
resource "aws_ecr_repository" "example_app_repository" {
  name                 = "example-app"
  image_tag_mutability = "MUTABLE"
}
```

By default an ECR repository has immutable image tag, meaning that when we upload a container image with a given tag, we can never upload an image with the same tag again. We want to set this to mutable, because we need to store Docker images for layer caching[1] when building the application. More about that in 5.2.2.

The second important part is to tell ECR how long to retain older images. We can setup different *life cycle policies* to tell ECR when to delete an image. Our system is setup to retain the last 30 images with a version tag, and delete all untagged images 5.1.

[1] `https://medium.com/swlh/docker-caching-introduction-to-docker-layers-84f20c48060a`



| | | | |
|---|---|---|---|
| ☐ | v0.0.15-179 | Image | 06. května 2022, 09:41:32 (UTC+02) |
| ☐ | v0.0.15-178 | Image | 05. května 2022, 14:40:36 (UTC+02) |
| ☐ | modules | Image | 05. května 2022, 14:36:51 (UTC+02) |
| ☐ | v0.0.15-177 | Image | 04. května 2022, 12:40:11 (UTC+02) |
| ☐ | v0.0.15-176 | Image | 03. května 2022, 18:57:03 (UTC+02) |
| ☐ | v0.0.15-175 | Image | 03. května 2022, 17:06:18 (UTC+02) |
| ☐ | v0.0.15-174 | Image | 03. května 2022, 14:44:14 (UTC+02) |
| ☐ | v0.0.15-173 | Image | 03. května 2022, 11:06:48 (UTC+02) |
| ☐ | v0.0.15-172 | Image | 03. května 2022, 09:10:05 (UTC+02) |

**Figure 5.1:** ECR Repository with tagged images

**IAM Policies and Roles.** Usually we want to keep IAM Policies and Roles tightly coupled with the resources, that require them, however there are some specific reusable policies, that we need in different modules and can define here without the risk of creating security threats. These policies are internal and are never attached to a specific user. For example here we define the *ECS task execution policy and role* 5.1, which is passed to individual service definitions, so ECS can run tasks by using *task definitions*.

**Source Code 5.1:** Task Execution Policy and Role

```
// ECS execution IAM role
resource "aws_iam_role" "ecs_task_execution" {
  name               = "ecs_task_execution"
  assume_role_policy =
   ↪  data.aws_iam_policy_document.assume_role_policy.json

}
resource "aws_iam_role_policy_attachment"
 ↪  "ecs_task_execution_policy" {
  role       = aws_iam_role.ecs_task_execution.name
  policy_arn = "arn:aws:iam::aws:policy/service-role/ \
               AmazonECSTaskExecutionRolePolicy"
}
```

Last but not least we create common S3 buckets in the global modules for *deploy scripts*, and *logs*.

### 5.1.2 Environment Specific Modules

The modules that are specific to each environment are located in:

```
infrastructure/provisioning/modules
```

We structure them logically into five folders according to the areas they relate to. Let's look at these directories in more detail:

**Cluster.** The cluster package defines how to setup Elastic Container Service. It tells ECS to use Fargate as a *capacity provider*[2] and sets up logging of services defined by ECS.

In section 3.2 we wrote that we chose ECS because it is easy to setup. By executing this small Terraform code 5.2 we have setup ECS with logging to CloudWatch.

**Source Code 5.2:** Elastic Container Service Setup

```
resource "aws_ecs_cluster" "ecs-cluster" {
 name = "${var.environment}-cluster"
 capacity_providers = [ "FARGATE", "FARGATE_SPOT" ]
 configuration {
  execute_command_configuration {
   logging    = "OVERRIDE"
   log_configuration {
    cloud_watch_log_group_name = var.cw_group_name
  }
 }
}
```

**Data stores.** The data-store package is split into two sub-modules: `RDS` and `Redis`. We mentioned that our system currently supports these two types of data storage in 4.4. However it is quite simple to extend the system to support a different data storage for example DynamoDB, or Elasticsearch, by creating a new sub-module in this directory.

Of course we keep in mind monitoring and logging of these resources. We also handle backups by creating a policy to keep the last seven days of RDS in snapshots, so we can easily restore the database to a specific point in time.

It is worth mentioning, that we do not allow public access to data storage, so it is impossible to connect to your database instance from the internet. However when using the system in practice we found out this restriction is only viable in production as the developers frequently want to modify data while testing the service they are developing. We setup RDS to be accessible within the VPC in environments other then production. This allows us to create a *bastion host*[3]. We allow SSH access with authorised keys to the

[2] `https://docs.aws.amazon.com/AmazonECS/latest/developerguide/cluster-capacity-providers.html`

[3] `https://en.wikipedia.org/wiki/Bastion_host`

bastion from the internet and that let's our developers connect to RDS with SSH port forwarding[4].

**Network.** The network package, as its name suggests, handles everything regarding networking:

- **Application Load Balancer** - The ALB sub-module creates the load balancer and sets up *target groups* to allow for routing network traffic to different microservices. We create listeners that handle traffic according to the subdomain in Host Header 5.3

  **Source Code 5.3:** ALB listener rule

  ```
  resource "aws_lb_listener_rule" "example_app_listener_rule"
   ↪ {
    listener_arn = aws_lb_listener.https_listener.arn
    action {
      type             = "forward"
      target_group_arn =
       ↪  aws_lb_target_group.example_app_target.arn
    }
    condition {
      host_header {
        values = ["example-app.${var.domain_name}*"]
      }
    }
  }
  ```

  However creating listeners on its own does not do anything, we attach them to a target group and then we need to specify what targets are linked to this target group. This is done from the other end, in service definition we define that a service belongs to this target group. In 5.4 we show a small part of service definition, that links it to target group and load balancer.

  **Source Code 5.4:** Link Load Balancer to Service

  ```
  resource "aws_ecs_service" "example_app_service" {
    // ...(truncated)
    load_balancer {
      // Referencing target group
      target_group_arn = var.target_group_arn
    }
  }
  ```

  For security reasons we also redirect all HTTP calls from port 80 to the secure HTTPS on port 443 with a simple redirect listener 5.5

  **Source Code 5.5:** Redirect HTTP to HTTPS

[4] https://www.ssh.com/academy/ssh/tunneling/example

```
resource "aws_lb_listener" "listener_http_forward" {
  load_balancer_arn = aws_alb.application_load_balancer.arn
  port              = "80"
  protocol          = "HTTP"
  default_action {
    type = "redirect"
    redirect {
      port        = "443"
      protocol    = "HTTPS"
      status_code = "HTTP_301"
    }
  }
}
```

- **Virtual Private Cloud** - This sub-module handles the creation of VPC, route tables, and subnets for RDS, ElastiCache and all private or public subnets we will need in the system. As this process is fairly straight forward and similar to most infrastructure VPCs, we were able to use an official HashiCorp module for AWS VPC [5].
- **Security Groups** - The security-groups sub-module handles security for east-west and north-south traffic. For example, here we specify, that the RDS instance can be accessed only from within VPC, and allow access only on port 5432 which is the default PostgreSQL port.

**Secrets and Environment Variables.** The secrets module specifies a base module which only specifies that we want to use parameter store. It accepts two main variables: `string_parameters` to define environment variables that do not have to be encrypted and `securestring_parameters`, which do get encrypted. We use this module with Terragrunt for each service that needs environment variables 5.6.

**Source Code 5.6:** Terragrunt Secrets and Environment Variables

```
terraform {
  source = "/modules/secrets//base-secrets"
}

inputs = {
  app_name = "example-app",
  securestring_parameters = [
    "AWS_REGION",
    "AWS_ACCESS_KEY_ID",
    "AWS_SECRET_ACCESS_KEY",
    "POSTGRES_PASS"
  ]
  string_parameters = [
```

[5]https://registry.terraform.io/providers/hashicorp/aws/latest

```
    "SERVER_PORT",
    "POSTGRES_DB",
    "POSTGRES_HOST",
    "POSTGRES_PORT",
    "POSTGRES_USER",
    "REDIS_HOST",
    "REDIS_PORT"
  ]
}
```

Outputs of this module are the final Amazon Resource Names(ARN) of each parameter. We use these outputs in service definitions to specify, which parameters to read on service creation.

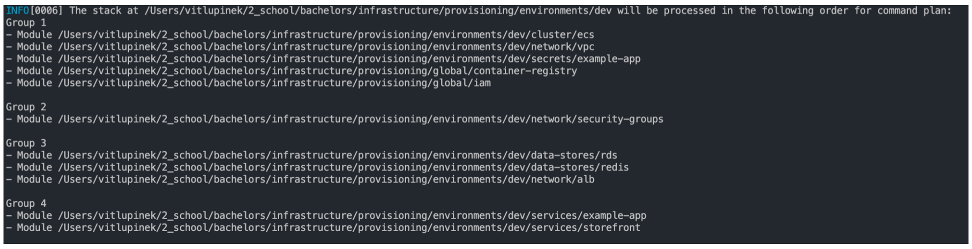
**Services.** The services module began as a place to define all microservices needed in the application, however as the development progressed we were able to generalise most of the necessary resources created by this module. Now this module defines only one base service sub-module, which creates task definitions, service definitions, logging and accepts the Task Execution role from 5.1. This module is then used by Terragrunt configuration for different services and environments. We pass different container images and environment variables but the core of the module remains the same.

### 5.1.3 Environments

The environments directory consists of all the *terragrunt.hcl* configuration files. Each environment has its own sub-directory. It closely resembles the structure of the `modules` directory. Here we specify all the services that should be deployed, configure data storages and specify what environment variables to create.

Each module can be provisioned on its own by entering the directory and executing the `terragrunt apply` command. This will first look at all the dependencies and check if they have been previously applied, if not, the command fails, otherwise Terragrunt will try to provision the module with specific configuration.

However if we wanted to provision the entire environment, we can enter the root directory for a given environment, and run `terragrunt run-all apply`. This command will go through all the *terragrunt.hcl* configuration files and by looking at the dependencies, it will create a sequence in which it applies individual modules. We can see this process in figure 5.2.



```
INFO[0006] The stack at /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev will be processed in the following order for command plan:
Group 1
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev/cluster/ecs
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev/network/vpc
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev/secrets/example-app
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/global/container-registry
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/global/iam

Group 2
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev/network/security-groups

Group 3
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev/data-stores/rds
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev/data-stores/redis
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev/network/alb

Group 4
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev/services/example-app
- Module /Users/vitlupinek/2_school/bachelors/infrastructure/provisioning/environments/dev/services/storefront
```

**Figure 5.2:** Terragrunt plan in run all mode

This process let's us create entire environments in minutes, and it is the simplicity we tried to achieve with our system.

**Pre-configured environments.** Out of the box we provide the users with three environments: Dev, Stage and Production. The users are able to create new environments simply by copying any of the environment folders, and changing the name of this environment in root *terragrunt.hcl*:

```
locals {
  region      = "eu-central-1"
  profile     = "example-profile"
  environment = "user-acceptance" //Name of the new
  ↪   environment
}
```

Now by executing `terragrunt run-all apply` in this newly created directory, we can deploy the whole environment infrastructure.

## 5.2 GitLab CI/CD pipelines

This section describes how we implemented the steps from 4.7. We usually define the configuration in a `.gitlab-ci.yml` file in the root directory.

All of our infrastructure code is located in a single repository, which is separated from the microservices repositories. We will use the terms *infrastructure* repository and *project* repository, to differentiate between them.

### 5.2.1 Extending Pipeline Jobs

The configuration in `.gitlab-ci.yml` file is automatically picked up by GitLab Runner on every push to version control, however this is still only a YAML file, and we can use multiple files to structure the pipelines.

GitLab let's us include configuration from different YAML files with the `include` keyword[6]:

```
include:
  - local: .gitlab/templates/common.yaml
```

[6] `https://docs.gitlab.com/ee/ci/yaml/includes.html`

We use this to define common pipeline configurations in one file and then reuse them in multiple files. Internally including files works simply as merging two configuration files together, but this is done by GitLab for us.

**Hidden pipeline jobs.** When defining a pipeline job we can prefix its name with a period to tag it as a hidden job:

```
.variables:
  variables:
    DOCKER_HOST: tcp://docker:2375
    DOCKER_DRIVER: overlay2
    LOG_COLOR_DEFAULT: \e[0m
    LOG_COLOR_DEPLOYER: \e[1;94m
    LOG_COLOR_COMMANDS: \e[1;32m
    DEPLOYER_DIR: ./tmp
    PRIVATE_TOKEN: $CI_JOB_TOKEN
    SCRIPTS_DIR: .gitlab/scripts
```

Then we can extend this hidden job definition in other jobs with the keyword `extends`:

```
.build:
  extends: .variables
  variables:
    SCRIPT_NAME: build.sh
```

We use this frequently throughout our code to keep the configuration DRY. However extending jobs works again by merging the jobs together, and if we define one part in the parent and child job, the configuration from parent will get overridden. To use only a part of the jobs definition we use the `!reference` tag:

```
.base:
  rules:
    - !reference [.needs_aws_vars_only_tags, rules]
```

This tells the runner, that we want to use only the `rules` from `.needs_aws_vars_only_tags` job.

## 5.2.2 Creating Docker Images

Building Docker images consists of executing a set of instructions defined in Dockerfile. Each instruction adds a layer to the final image which gets cached for future builds of the image. In our system, we want to use Docker to develop the application, test it, and then run it in production. However in the production image we do not need any third party dependencies used only for development, and we want to keep the size of this image as small as possible. To manage multiple different images, and keep the setup DRY, we can leverage from the multi-stage build functionality[7].

[7] https://docs.docker.com/develop/develop-images/multistage-build/

### Multi-stage builds

Our system works with four core stages and one optional but useful stage for dependencies:

- **deps** - This optional and simple stage extracts dependencies, and scripts from the `package.json` file. We do this to prevent invalidating layer cache of Docker. Because `package.json` keeps the version number of applications, the contents of this file will change frequently, but only in this version number, however when a file changes for Docker it means something is different and it tries to rebuild the whole stage. By keeping only dependencies, and scripts in the `package.json` file, we can cache dependency installation in the next stage.

  **Source Code 5.7:** Package json stage

  ```
  FROM alpine AS deps

  COPY package.json /var

  RUN apk add jq

  RUN jq '{ dependencies, devDependencies, workspaces,
  ↪  private, scripts }' < /var/package.json >
  ↪  /tmp/deps.json
  ```

- **dev_modules** - This image stage consists of all development dependencies, all source code, and is quite big in size. Our Dockerfile for Node.js looks as follows:

  **Source Code 5.8:** Development dependencies stage

  ```
  # ---------- DEV DEPENDENCIES ----------
  FROM node:14-alpine AS dev_modules_image

  WORKDIR /usr/src/app

  RUN apk add curl git
  RUN apk add --virtual .build-deps ca-certificates wget

  COPY --from=deps /tmp/deps.json ./package.json
  COPY yarn.lock ./
  RUN yarn install
  ```

  We can notice the `COPY --from=deps` part which handles file transfer between stages. Now if we build an image targeting this `dev_modules_image`, we can use it for development purposes.

- **build** - The build stage copies installed dependencies from `dev_modules_image`, then copies all source code files, and finally builds the application into a `dist/` folder, which can be used in the `runtime_image`.

**Source Code 5.9:** Build stage

```
FROM node:14-alpine AS build_image

WORKDIR /usr/src/build

COPY --from=dev_modules_image /app/node_modules
 ↪  ./node_modules
COPY package.json ./
COPY tsconfig.json tsconfig.build.json ./
COPY src ./src

RUN yarn build
```

- **modules** - The modules stage extends the previously defined *dev_modules* stage, and removes dependencies needed only in development.

**Source Code 5.10:** Modules stage

```
FROM dev_modules_image as modules_image
WORKDIR /usr/src/app
ENV NODE_ENV=production
RUN yarn install --production
```

- **latest or runtime_image** - This is the final stage, which is used in ECS task definitions. It combines the previous stages into one by copying only the necessary files. This decreases the final size of the image, because we do not store for example Yarn[8] cache in this image.

**Source Code 5.11:** Runtime Docker Image stage

```
FROM node:14-alpine as runtime_image

ENV NODE_ENV=production

WORKDIR /app

COPY --from=modules_image /app/node_modules/
 ↪  ./node_modules/
COPY --from=build_image /usr/src/build/dist/ ./dist/
COPY package.json ./package.json

USER node
```

## Docker layer caching

We stated that each instruction in Dockerfile is a layer which can be cached. Docker creates hashes for each layer from the related resources, and if a new

[8] https://yarnpkg.com/

hash matches the old hash it uses the layer from cache. This can greatly speed up the build process of images. However each pipeline job is run in a new environment, so the cache here is clean and everything needs to be rebuild from scratch. We can specify a tagged docker image to be used as a cache source by using the `--cache-from` argument, which lets us leverage layer caching. We do this in our code:

**Source Code 5.12:** Docker layer caching in CI/CD

```
docker build --target runtime_image \
    --cache-from $DOCKER_REPOSITORY:dev-modules \
    --cache-from $DOCKER_REPOSITORY:modules \
    --cache-from $DOCKER_REPOSITORY:build \
    --cache-from $DOCKER_REPOSITORY:latest \
    -t $DOCKER_REPOSITORY:latest .
```

Here we use all four modules as cache sources for the fastest build time.

### 5.2.3 Docker Images in pipelines

Building images in our system is a little bit more complex then usual as we are using the multi-stage builds. And because we want to use layer caching, we need to first build images in individual stages and upload them to ECR. This is done by the `docker push` command. When we have images for different stages uploaded on ECR, we can then download them in the next run of our pipeline, by using `docker pull`.

The process of building new image includes these four steps:

- Pull latest images from ECR, so we can use them with the `--cache-from` argument.
- Using these images as cache sources, rebuild them one at a time
- Tag the newly created images.
- Upload the images to ECR, so we can use them in next pipeline, and as a source in ECS service definitions.

The function that builds the runtime image using cache sources looks as follows:

**Source Code 5.13:** Build runtime Docker image script

```
pull_latest () {
  # Pull all the images for caching
  docker pull $DOCKER_REPOSITORY:dev-modules
  docker pull $DOCKER_REPOSITORY:modules
  docker pull $DOCKER_REPOSITORY:build
  docker pull $DOCKER_REPOSITORY:latest
}
```

```
build_push_runtime_stage_image() {
  # Use previous images as cache source
  docker build --target runtime_image \
    --cache-from $DOCKER_REPOSITORY:dev-modules \
    --cache-from $DOCKER_REPOSITORY:modules \
    --cache-from $DOCKER_REPOSITORY:build \
    --cache-from $DOCKER_REPOSITORY:latest \
    -f $DOCKERFILE_PATH \
    -t $DOCKER_REPOSITORY:$CI_BUILD_REF_NAME \
    -t $DOCKER_REPOSITORY:latest .
}
```

The script in `infrastructure/.gitlab/scripts/build.sh` has also other commands which are mostly for logging purposes. However the script also provides functions for creating other images such as the *build stage* image.

The functions 5.13, are then called from the GitLab CI/CD configuration.

**Source Code 5.14:** Build runtime Docker image job

```
.build:
  extends: .base
  variables:
    SCRIPT_NAME: build.sh

  script:
    - !reference [.fetch_commander_script, script]
    - commander pull_latest
    - commander build_push_build_stage_image
    - commander build_push_runtime_stage_image
```

### 5.2.4 Code Quality and Linting

The system's first step after a developer pushes code into version control is to check the quality of code and apply lint rules. We can execute these two jobs in parallel in one pipeline stage.

To check the quality of code, we used GitLab's predefined docker image for code quality 5.15, which generates a JSON file with all problems in the code. Then we store this file for later use. However even if we find *code smells* or other issues in the code, the pipeline does not get interrupted.

**Source Code 5.15:** Code quality check

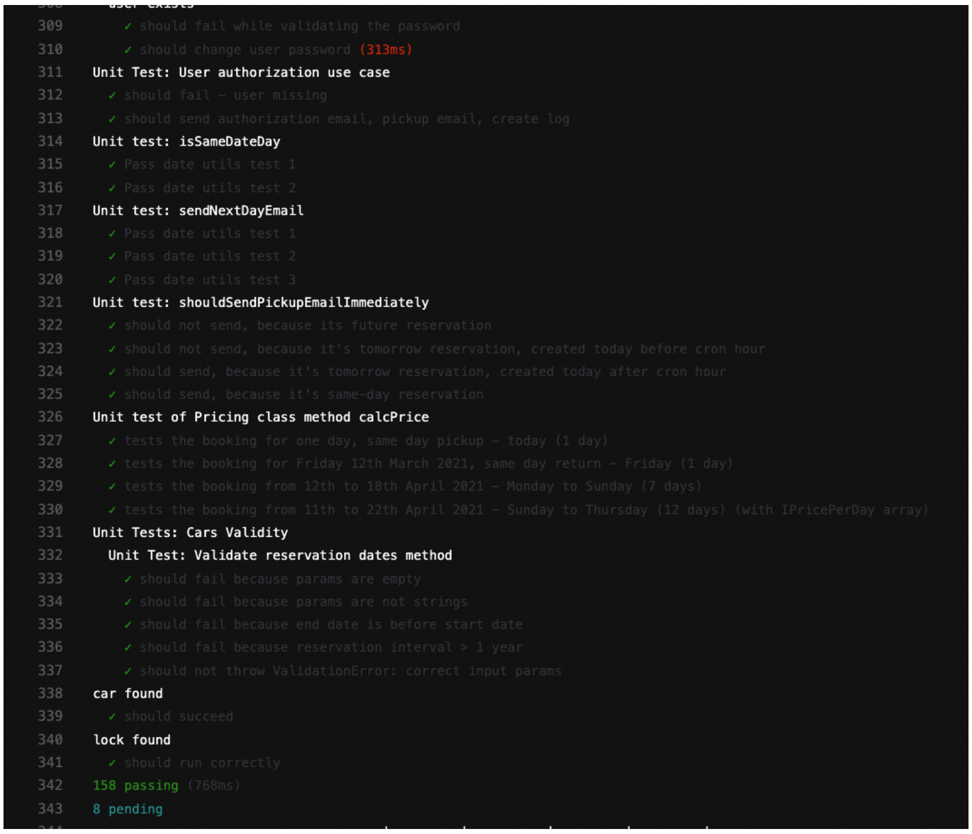
```
DOCKER_IMAGE="registry.gitlab.com/gitlab-org/ci-cd/codequality"
docker run \
    --env SOURCE_CODE="${SOURCE_CODE:-$PWD}" \
    --volume "${SOURCE_CODE:-$PWD}":/code \
    --volume /var/run/docker.sock:/var/run/docker.sock \
    ${DOCKER_IMAGE}:${CODE_QUALITY_VERSION:-latest} /code
```

The second part is linting the code. We can use the previously created Docker image with dev dependencies `dev_modules_image` and execute the command `yarn lint`. This process is specific to Node.js applications and in the future it would be better to create specific Docker images that run a lint command at runtime.

### 5.2.5 Automated Testing

Running automated tests is an essential part of the development process. In this stage we can automatically find issues with the final application, and there is no need to trouble testers. Again we use docker images to run tests of the application, because it allows the environment to look similar to the environment the application will actually run in.

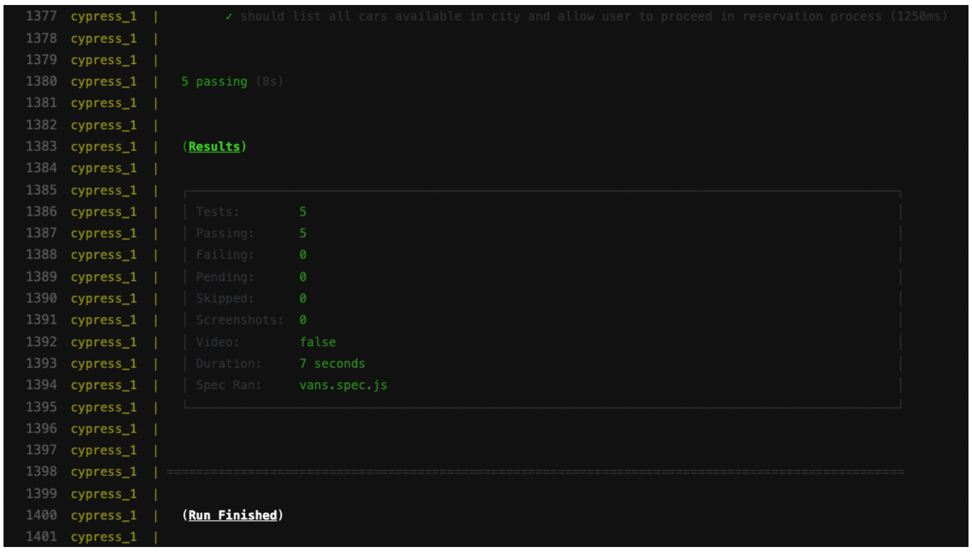
We have setup steps for *unit* testing as well as *end-to-end testing*, however it would be simple to extend these steps for other testing strategies such as *integration*. Figure 5.3 shows an example of the system executing unit tests in a pipeline job and figure 5.4 shows an example of running end-to-end tests with cypress[9].



```
      ✓ should fail while validating the password
      ✓ should change user password (313ms)
  Unit Test: User authorization use case
    ✓ should fail - user missing
    ✓ should send authorization email, pickup email, create log
  Unit test: isSameDateDay
    ✓ Pass date utils test 1
    ✓ Pass date utils test 2
  Unit test: sendNextDayEmail
    ✓ Pass date utils test 1
    ✓ Pass date utils test 2
    ✓ Pass date utils test 3
  Unit test: shouldSendPickupEmailImmediately
    ✓ should not send, because its future reservation
    ✓ should not send, because it's tomorrow reservation, created today before cron hour
    ✓ should send, because it's tomorrow reservation, created today after cron hour
    ✓ should send, because it's same-day reservation
  Unit test of Pricing class method calcPrice
    ✓ tests the booking for one day, same day pickup - today (1 day)
    ✓ tests the booking for Friday 12th March 2021, same day return - Friday (1 day)
    ✓ tests the booking from 12th to 18th April 2021 - Monday to Sunday (7 days)
    ✓ tests the booking from 11th to 22th April 2021 - Sunday to Thursday (12 days) (with IPricePerDay array)
  Unit Tests: Cars Validity
    Unit Test: Validate reservation dates method
      ✓ should fail because params are empty
      ✓ should fail because params are not strings
      ✓ should fail because end date is before start date
      ✓ should fail because reservation interval > 1 year
      ✓ should not throw ValidationError: correct input params
  car found
    ✓ should succeed
  lock found
    ✓ should run correctly
  158 passing (768ms)
  8 pending
```

**Figure 5.3:** Pipeline job executing unit tests.

[9] `https://www.cypress.io/`



```
cypress_1  |        ✓ should list all cars available in city and allow user to proceed in reservation process (1250ms)
cypress_1  |
cypress_1  |
cypress_1  |   5 passing (8s)
cypress_1  |
cypress_1  |
cypress_1  |   (Results)
cypress_1  |
cypress_1  |   ┌──────────────────────────────────────────────────────────────────────────────────────────────┐
cypress_1  |   │ Tests:        5                                                                               │
cypress_1  |   │ Passing:      5                                                                               │
cypress_1  |   │ Failing:      0                                                                               │
cypress_1  |   │ Pending:      0                                                                               │
cypress_1  |   │ Skipped:      0                                                                               │
cypress_1  |   │ Screenshots:  0                                                                               │
cypress_1  |   │ Video:        false                                                                           │
cypress_1  |   │ Duration:     7 seconds                                                                       │
cypress_1  |   │ Spec Ran:     vans.spec.js                                                                    │
cypress_1  |   └──────────────────────────────────────────────────────────────────────────────────────────────┘
cypress_1  |
cypress_1  |
cypress_1  | ==================================================================================================
cypress_1  |
cypress_1  |   (Run Finished)
cypress_1  |
```

**Figure 5.4:** Pipeline job executing e2e tests.

Any failing tests will interrupt the pipeline and alert developers, that there is a problem that needs to be solved before we can build their application.

### 5.2.6 Deploying to Cloud Environment and Launching New Tasks

After we have built the final runtime image, we can finally proceed to the last step which is deploying the application to an environment. We allow developers to trigger this stage manually, as we want to let them decide when a new version of application gets pushed to an environment.

The deployment process is done by Terragrunt. First we need to update the task definition with a new image tag. And then we need to apply it to our infrastructure 5.16.

**Source Code 5.16:** Deploy service template

```
.deploy-service:
  stage: deploy
  image: alpine/terragrunt
  variables:
    VERSION: "latest"
    ENVIRONMENT: "dev"
  rules:
    - when: never
  allow_failure: false
  script:
    - cd
    ↪  provisioning/environments/$ENVIRONMENT/services/$APP_NAME
    - export TF_VAR_image_tag=$VERSION
    - terragrunt init
    - terragrunt validate
    - terragrunt plan
    - terragrunt apply --auto-approve
```

This is a base job definition, which we can then extend and pass variables: `APP_NAME`, to specify which service to deploy, `ENVIRONMENT`, to specify which environment to deploy to, and `VERSION`, which is the new Docker image tag.

However the Terragrunt code is located in our *infrastructure* repository, but when running pipelines the context is in *project* repository. To avoid copying all files between repositories, and creating potential security risks, we can remotely trigger a pipeline in *infrastructure* repository from the *project* repository. This is done with the `trigger` keyword 5.17, and can be used on public repositories, and repositories in the same group.

**Source Code 5.17:** Trigger remote pipeline

```
.deploy-base:
    stage: deploy
    variables:
        <<: *globals
        ENVIRONMENT: "dev"
    trigger:
        project: git-group/infrastructure
        branch: infrastructure
    only:
        - tags
    when: manual
```

This produces a *downstream* pipeline setup in *project* repository, which is shown in figure 5.5. The *downstream* job is executed in *infrastructure* repository.

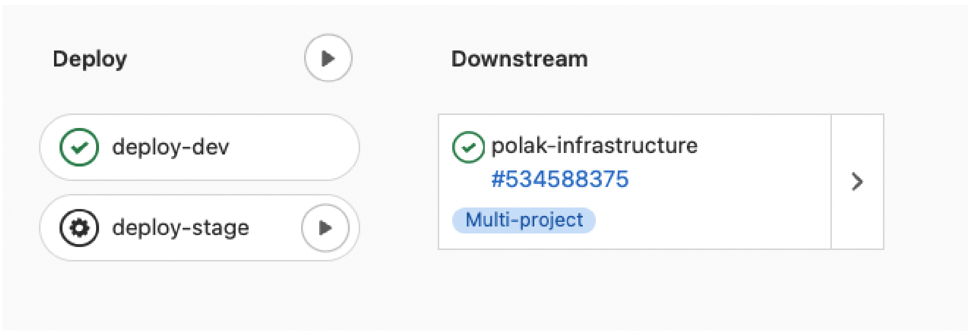


**Figure 5.5:** Remote pipeline trigger

# Chapter 6
# Example Project

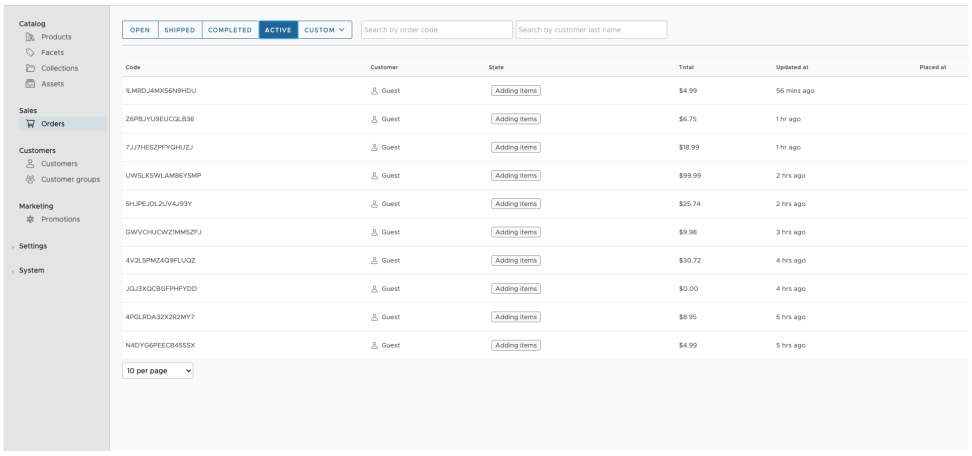
The system was tested on a simple example E-commerce project using a template called Vendure[1]. This project is an open source headless E-commerce platform, meaning it does not have a specific client, or storefront application, but it only provides a GraphQL[2] interface. For the frontend application, we will use a *demo-storefront*[3] application, that has implementation for Vendure backend interface. The code for this application is located in `project/storefront` and `project/example-app` folders.

## 6.1 Vendure E-commerce

This application consists of two Node.js services. The backend application, that handles all logic with data, communicates with a relational database, and stores request session data in Redis. The second part is an Admin UI component for managing product catalogue, handling orders, and other functions needed to run an online shop. The application runs a web server, that handles HTTP requests, so we can create a microservice setup for it 5.1.2. The database is a PostgreSQL running in a RDS instance.

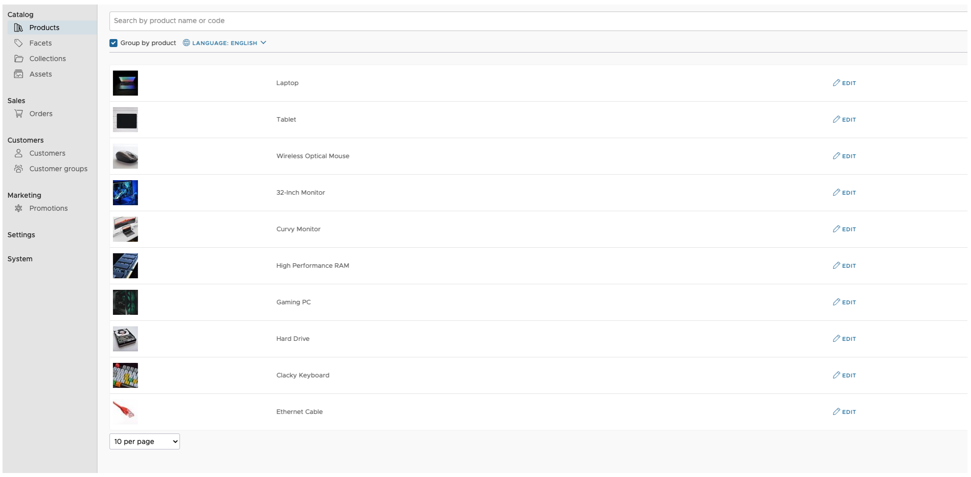
A preview of the Admin UI component is shown in figures 6.1, 6.3, 6.2.

[1]`https://www.vendure.io/`
[2]`https://graphql.org/`
[3]`https://github.com/vendure-ecommerce/storefront`



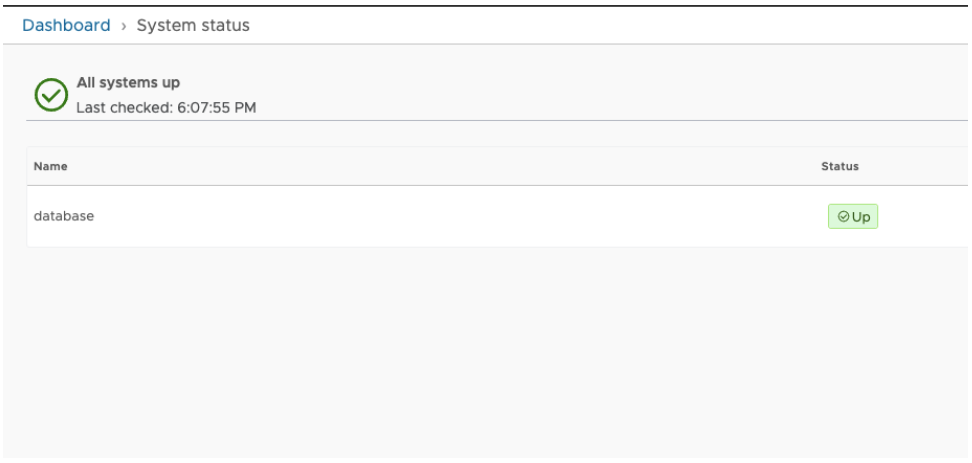
| Code | Customer | State | Total | Updated at | Placed at |
|---|---|---|---|---|---|
| 1LMRDJ4MXS6N9HDU | Guest | Adding items | $4.99 | 56 mins ago | |
| Z6PBJYU9EUCQLB36 | Guest | Adding items | $6.75 | 1 hr ago | |
| 7JJ7HESZPFYQHUZJ | Guest | Adding items | $18.99 | 1 hr ago | |
| UWSLKSWLAM86YSMP | Guest | Adding items | $99.99 | 2 hrs ago | |
| 5HJPEJDL2UV4J93Y | Guest | Adding items | $25.74 | 2 hrs ago | |
| GWVCHUCWZ1MM5ZFJ | Guest | Adding items | $9.98 | 3 hrs ago | |
| 4V2L5PMZ4Q9FLUQZ | Guest | Adding items | $30.72 | 4 hrs ago | |
| JQJ3XQCBGFPHFYDD | Guest | Adding items | $0.00 | 4 hrs ago | |
| 4PGLRDA32X2R2MY7 | Guest | Adding items | $8.95 | 5 hrs ago | |
| N4DYG6PEECB45SSX | Guest | Adding items | $4.99 | 5 hrs ago | |

**Figure 6.1:** Orders tab



| | | |
|---|---|---|
| | Laptop | EDIT |
| | Tablet | EDIT |
| | Wireless Optical Mouse | EDIT |
| | 32-Inch Monitor | EDIT |
| | Curvy Monitor | EDIT |
| | High Performance RAM | EDIT |
| | Gaming PC | EDIT |
| | Hard Drive | EDIT |
| | Clacky Keyboard | EDIT |
| | Ethernet Cable | EDIT |

**Figure 6.2:** Catalogue tab



All systems up
Last checked: 6:07:55 PM

| Name | Status |
|---|---|
| database | Up |

**Figure 6.3:** System monitor tab

The backend application exposes only a GraphQL interface, which can be accessed through a playground[4].

## 6.2 Demo storefront

The storefront application is the final web page, that customers access. It provides product listings, filtering, searching, and then adding items to cart and proceeds with checkout to create orders. Again for illustration figures 6.4 and 6.5

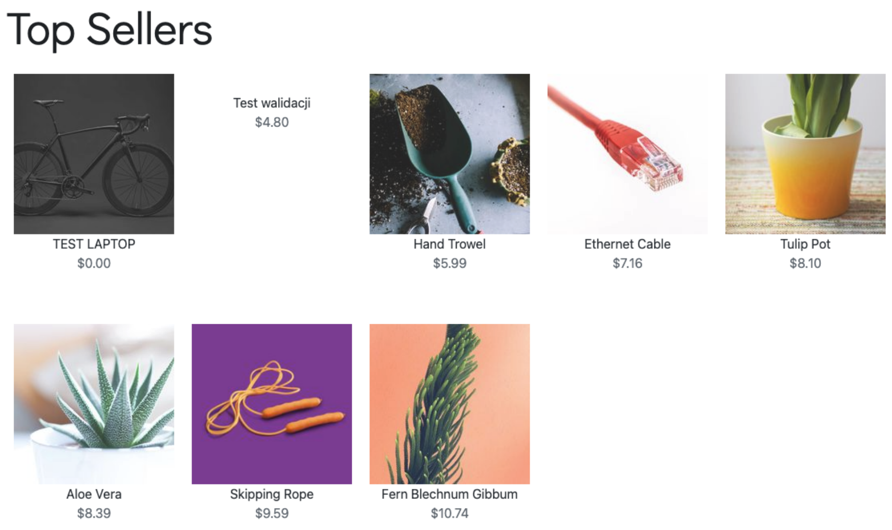


**Figure 6.4:** Homepage top sellers

[4]https://github.com/graphql/graphql-playground

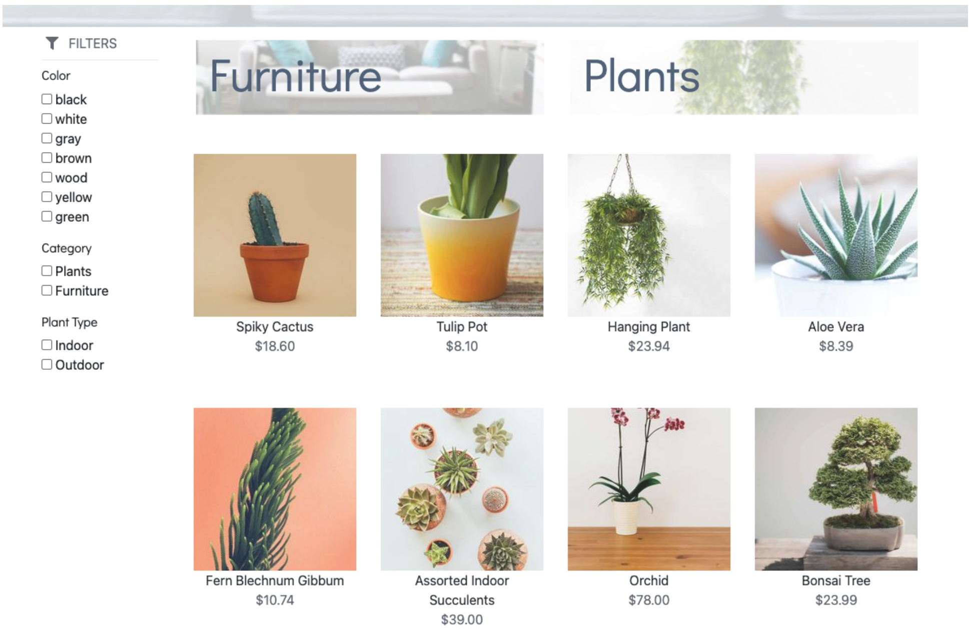


**Figure 6.5:** Product Listings

## 6.3 CI/CD and Infrastructure

In figure 6.6 we demonstrate the pipeline stages for the system, that get created after a tagged commit is pushed to GitLab.

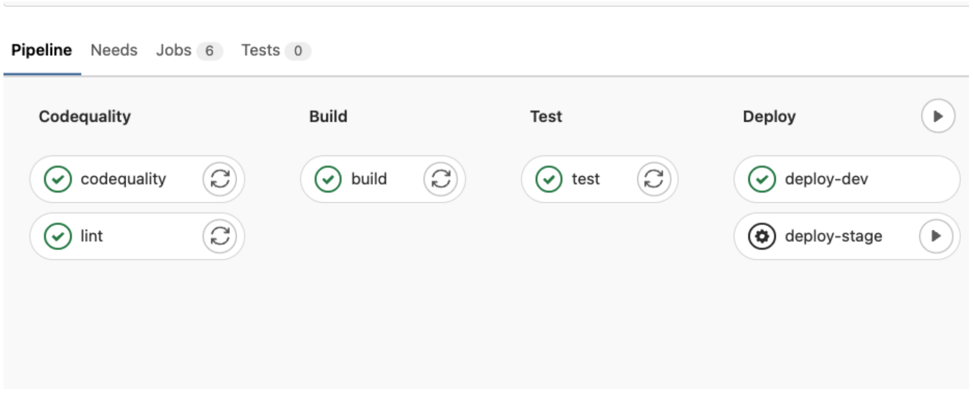
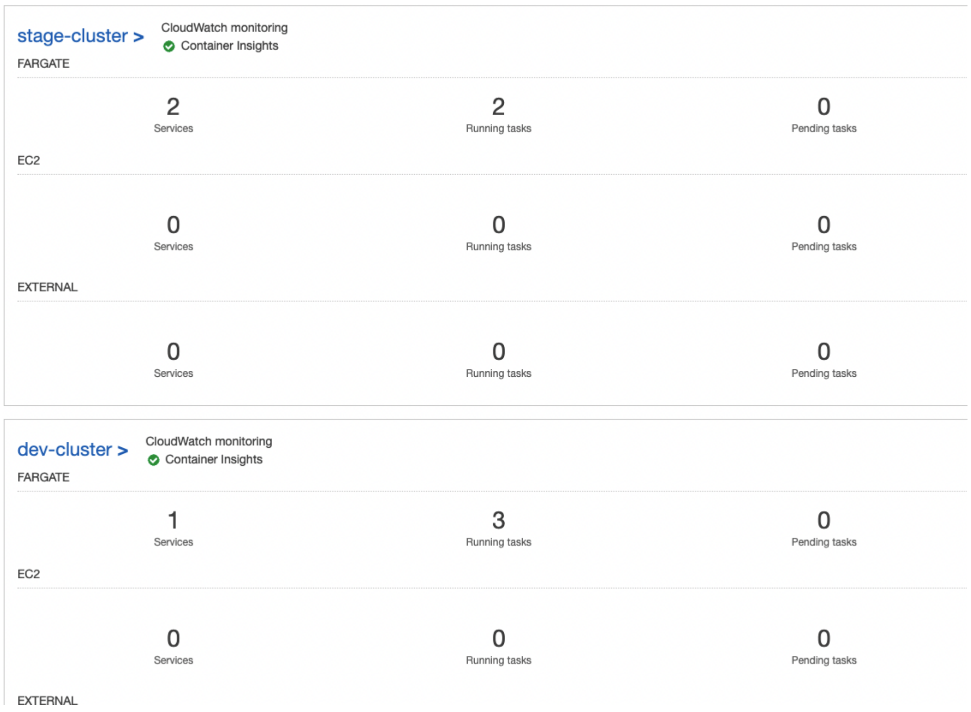


**Figure 6.6:** CI/CD Pipeline

We tested the system by creating *dev* and *stage* environments. In figure 6.7 we can see two ECS clusters for these stages. From the figure we can see, that we scaled the backend service to run three tasks, and the load balancer in front of it will distribute work among these tasks evenly.


stage-cluster >

CloudWatch monitoring

✓ Container Insights

FARGATE

| 2 | 2 | 0 |
|---|---|---|
| Services | Running tasks | Pending tasks |

EC2

| 0 | 0 | 0 |
|---|---|---|
| Services | Running tasks | Pending tasks |

EXTERNAL

| 0 | 0 | 0 |
|---|---|---|
| Services | Running tasks | Pending tasks |

dev-cluster >

CloudWatch monitoring

✓ Container Insights

FARGATE

| 1 | 3 | 0 |
|---|---|---|
| Services | Running tasks | Pending tasks |

EC2

| 0 | 0 | 0 |
|---|---|---|
| Services | Running tasks | Pending tasks |

EXTERNAL

**Figure 6.7:** Dev and Stage ECS clusters

# Chapter 7
# Conclusion

The goal of this thesis was to design and implement an easy to use system of configuration files that provide software development teams with out of the box Continuous Integration and Continuous Delivery pipelines, as well as a convenient way of creating infrastructure for cloud applications. Before we started the implementation, we analysed the challenges the development teams face while implementing applications. We also discussed ways to solve these issues with automation by introducing Infrastructure as a Code and CI/CD to the process.

We introduced the technology stack used in the system and some of the benefits of using tools like Terraform, Terragrunt or GitLab CI/CD. In the second half of the thesis we proposed a design of the implemented system, and went through the implementation, describing some of the challenges we faced in the process. To validate functionality of our system, we also tested the system on a example E-commerce project.

The result of this thesis is a reusable system consisting of two parts: IaC implemented in Terraform with the Terragrunt wrapper, and the CI/CD part of the system which leverages from the GitLab CI/CD tool configured by YAML files.

As of May 2022 the system is already being used in practice to support development of a web application for 3D modelling of workshop furniture. As the development process continues, we get valuable feedback from actual software developers, which we can address in the future 7.1. Another milestone for our system will be the deployment to production environment, when the application will be served to end users.

## 7.1 Further work

Throughout chapter 4 we already mentioned some nice to have features for the system including automatic creation of metrics, automatic creation of parameters from version control, and more generic CI/CD jobs that rely on the definition in Dockerfile. However the most requested feature from developers was to create alerts for situations such as successful or failed deployment of new application version.

# Bibliography

[AN20] *Using ansible to configure nginx on ubuntu and host a static website* `https://graspingtech.com/ansible-nginx-static-site/`, August 2020.

[Awa] Rahul Awati, *How load balancers work.* `https://www.techtarget.com/searchaws/definition/application-load-balancer`.

[BKH21] Florian Beetz, Anja Kammer, and Simon Harrer, *Gitops - cloud-native continuous deployment* `https://leanpub.com/gitops/`, July 2021.

[Lou12] Mike Loukides, *What is devops?* `http://radar.oreilly.com/2012/06/what-is-devops.html`, June 2012.

[NEU16] NEUVECTOR, *Difference between east-west and north-south traffic.* `https://blog.neuvector.com/article/securing-east-west-traffic-in-container-based-data-center`, October 2016.

[Ora] Oracle, *Managing user sessions.* `https://docs.oracle.com/cd/E19683-01/817-2172-10/dwsessn.html`.

[Sta20] John Starmer, *Difference between horizontal and vertical scaling.* `https://opsani.com/blog/scale-up-vs-scale-out-whats-the-difference/`, July 2020.

[Tea16] Nginx Team, *What is nginx?* `https://www.nginx.com/resources/glossary/nginx/`, May 2016.

# Appendix A

# Attachment Content

We split the attachment into `infrastructure` and `project` directories, however in practice they should be separate Git repositories.

```
root/
├── infrastructure/
│   ├── .gitlab/
│   │   ├── scripts/ - bash scripts that execute build, code lint,
│   │   │   testing and other tasks
│   │   └── templates/ - Gitlab CI/CD yaml files which are extended
│   │       in the final repository
│   ├── project-templates/ - Example Dockerfile and CI/CD extension
│   │   to use in projects
│   ├── provisioning/
│   │   ├── environments/ - Terragrunt configuration files for each
│   │   │   environment
│   │   ├── global/ - global Terraform modules that are shared among
│   │   │   environments
│   │   └── modules/ - environment specific modules, that are used
│   │       in Terragrunt configurations
│   └── .gitlab-ci.yml - pipeline configuration for this repository
└── project/
    ├── example-app/
    │   ├── src/ - server side source code and configuration
    │   ├── static/ - static files for admin user interface
    │   ├── .gitlab-ci.yml - Pipeline configuration for this repository
    │   └── Dockerfile - Dockerfile definition for this app
    └── storefront/
        ├── .gitlab-ci.yml - Pipeline configuration for this repository
        └── Dockerfile - Dockerfile definition for this app
```