Degree Final Project

Bachelor Degree in Informatics Engineering

Platform for deploying a highly available, secure and scalable web hosting architecture to the AWS cloud with Terraform

THESIS REPORT

June 27th, 2022

Autor: Martí Juncosa Palahí
Director: Llorenç Cerdà Alabern (AC)
Speciality: Information Technologies

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Resum

L'objectiu d'aquest projecte és la creació d'una plataforma capaç de realitzar tots els passos necessaris per tal de generar una infrastructura preparada per allotjar al núvol d'Amazon Web Services els portals web més exigents.

L'interès d'aquest servei rau en l'elevada disponibilitat que aquests llocs web requereixen, ja que no es poden permetre estar fora de servei i requereixen d'una seguretat extrema i un cost de manteniment elevat a causa de la demanda variable i impredictible que solen tenir.

La plataforma presentada, desenvolupada en Python, permet, per una banda, automatitzar la creació de la infraestructura gràcies al llenguatge Terraform, un dels pilars de la infraestructura com a codi (IaC), que permet crear, esborrar i destruir amb facilitat aquest tipus de desplegaments cloud. Per altra banda, prepara el codi per a ser mantingut en el temps de forma ràpida i eficient, tot potenciant el treball colaboratiu i en equip (CI/CD) gràcies al control de versions de GitHub i l'emmagatzematge del codi Terraform al núvol d'AWS.

Al llarg de la memòria, s'introdueixen els conceptes principals de Terraform, s'enumeren i expliquen en profunditat cadascun dels components que conformen la infraestructura i es detalla com s'ha construït la plataforma així com les accions que porta a terme.

Es conclou reflexionant sobre els avantatges que implica el cloud per a la indústria i els errors, temps i diners que estalvia fer-ne ús juntament amb un llenguatge d'infraestructura com a codi, així com els beneficis que aporta dissenyar des d'un bon inici entorns que facilitin el desenvolupament continu i el treball en equip.

Tot el codi desenvolupat durant el projecte es pot consultar en el següent repositori públic de GitHub: https://github.com/j1nc0/TFG [12]

Resumen

El objetivo de este proyecto es la creación de una plataforma capaz de realizar todos los pasos necesarios para generar una infraestructura preparada para alojar en la nube de Amazon Web Services los portales web más exigentes.

El interés de este servicio radica en la elevada disponibilidad que estos sitios web requieren, ya que no pueden permitirse estar fuera de servicio y requieren de una seguridad extrema y un coste de mantenimiento elevado debido a la demanda variable e impredecible que suelen tener.

La plataforma presentada, desarrollada en Python, permite, por un lado, automatizar la creación de la infraestructura gracias al lenguaje Terraform, uno de los pilares de la infraestructura como código (IaC), que permite crear, borrar y destruir con facilidad este tipo de despliegues cloud. Por otro lado, prepara el código para ser mantenido en el tiempo de forma rápida y eficiente, potenciando el trabajo colaborativo y en equipo (CI/CD) gracias al control de versiones de GitHub y el almacenamiento del código Terraform en la nube de AWS.

A lo largo de la memoria, se introducen los principales conceptos de Terraform, se enumeran y explican en profundidad cada uno de los componentes que conforman la infraestructura y se detalla cómo se ha construido la plataforma así como las acciones que lleva a cabo.

Se concluye reflexionando sobre las ventajas que implica el cloud para la industria y los errores, tiempo y dinero que ahorra su uso junto con un lenguaje de infraestructura como código, así como los beneficios que aporta diseñar desde un buen inicio entornos que faciliten el desarrollo continuo y el trabajo en equipo.

Todo el código desarrollado durante el proyecto se puede consultar en el siguiente repositorio público de GitHub: https://github.com/j1nc0/TFG [12]

Abstract

The goal of this project is to create a platform capable of performing all the necessary steps in order to generate an infrastructure ready to host the most demanding web portals in the Amazon Web Services cloud.

The interest of this service lies in the high availability that these websites require, as they cannot afford to be out of service and require extreme security and a high maintenance cost due to the variable and unpredictable demand that they usually have.

The platform presented, developed in Python, allows, on the one hand, to automate the creation of the infrastructure thanks to the Terraform language, one of the pillars of the infrastructure as a code (IaC), which allows you to easily create, delete and destroy this type of cloud deployments. On the other hand, it prepares the code to be maintained over time quickly and efficiently, enhancing teamwork (CI / CD) through GitHub version control and Terraform code storage in the AWS cloud.

Throughout the report, the main concepts of Terraform are introduced, each of the components that make up the infrastructure are listed and explained in depth, and it is detailed how the platform was built as well as the actions it carries out.

The thesis concludes by reflecting on the benefits of the cloud for the industry and the mistakes, time and money it saves to use it along with an infrastructure as a code language, as well as the benefits of designing from the beginning environments that facilitate continuous development and teamwork.

All the code developed during the project can be consulted in the following public GitHub repository: https://github.com/j1nc0/TFG [12]

Acknowledgments

I would like to thank the following people who helped me undertake this report:

My family, for supporting me during this intense months, but specially during the toughest times of the career. You all have made my life so much easier!

My thesis report director, Llorenç Cerdà, for accepting to provide guidance to the project I proposed and suggesting the best practices as well as really useful recommendations. Writing the project in Latex has turned out to be a huge improvement.

Finally, thanks to the FIB professors that have showed me that when I chose the Information Technologies speciality I was in the right path. This led me to find out cloud computing, which is what I like to do and my true vocation.

Contents

List of Figures

List of Tables

Chapter 1

Context and scope

1.1 Context

In this section I am going to introduce the reader to cloud computing and its benefits, I will define and explain the general terms and concepts of this project, I will identify the problem to be resolved and finally I will point out the possible stakeholders this project will have.

1.1.1 General contextualization

Cloud computing has been growing increasingly in popularity among each and every one IT company. It offers a wide variety of advantages compared to the traditional on premise infrastructure, especially when we talk about the Amazon Web Services cloud [3] (from now on AWS), the cloud provider I will use and the one that has more than 30% of the market share and offers more than 2000 services.

Those advantages are the following:

1. Anyone can go global in minutes: AWS has servers all over the world. And thus, latency is minimal.
2. Stop maintaining data centers, focus on building apps and software. All the infrastructure is maintained by AWS, you do not have to spend a single euro on that.
3. Benefit of massive economies of scale: cheaper prices and discounts on all the services offered. You pay less for better service.
4. Increase the speed and agility of your developments with the hundreds of services provided and innovate quicker.

5. Capacity is matched exactly to your demand. You do not have to provision the compute power or inbound traffic peaks. If you need more power, just scale out.

6. Trade capital expense for variable expense: you pay for what you use. Not upfront investments

7. The AWS power grid is 100% green. All the power comes from renewable sources.

Infrastructure as Code (IaC) [16] offers a way to configure all those services through code. The most recognized language is called Terraform [22], which perfectly integrates with AWS.

When you combine both of them, this easens the process of creating infrastructure, because once the code is written, you always have it and you do not have to repeat the processes you must do if you used the AWS web interface. It also eases the way to maintain it as you only need to modify the part of the code you want to change.

However, IT teams normally have multiple developers working at the same time improving and maintaining this code. This is where the Continuous Deployment / Continuous Integration pipelines and version control, (GitHub) have its role. They perform a series of steps in order to automatically ensure that the code uploaded is correct, functional and no one overrides the code of others.

By combining AWS, Terraform and GitHub [11] I want to create a platform that eases the creation of really complex web hosting infrastructure. Making something that not many years ago was tedious and time consuming almost instantly.

This project is perfect for a WordPress web page, however this does not mean that it is not suitable for other types of web page softwares and CMS. This project mainly focuses on the deployment of the necessary infrastructure on AWS, not on the configuration of the servers. Its intention is to be adaptable to the different software and configurations a web page could need.

1.1.2 Terms and concepts

- AWS: Amazon Web Services is a subsidiary of Amazon that provides on-demand cloud services and compute power to companies and individuals on a pay-as-you-go basis.

  These services provide abstract infrastructure and building tools. One of those services, and the one I will use the most is called EC2 (Elastic Compute Cloud), which offers computational power in the cloud via a cluster of servers.

  I will also make use of other popular services such as IAM (Identity Access Management) for roles and permissions among all services, RDS (Relational Database Service) for the database, Cloudwatch for monitoring and logging purposes and S3 (Simple Storage Service) for cloud storage purposes.

- IaC: Infrastructure as code is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.
- Terraform: Terraform is an open-source infrastructure as code software tool that is used to manage a large amount of cloud services. The Hashicorp team has developed plugins for the majority of cloud providers that allow users to manage an external API. Thanks to that I will be able to work with AWS.
- CI/CD [27]: In software development is the combined practice of Continuous Integration and Continuous Deployment. Which results in the frequent delivery of apps to customers by applying automation into the stages of development.

1.1.3 Problem to be resolved

The issue I am resolving with this project is the complexity involved in automating the creation of services in the cloud. With the platform, individuals with little to no knowledge about cloud services and even IaC could deploy and, what is more important, manage services in the AWS cloud. And what is more, they could do it faster than the average. All thanks to a fully automated process with a simple and intuitive interface.

The aim of this TFG is to design an infrastructure that can be used to launch high available-performant demanding web services, specially web pages that require a web hosting that is capable to assume different workloads, traffic spikes and is fault tolerant. Basically, build services that are available (percentage of operability), durable (are on until you decide to shut them down), resilient (can recover quickly from damage) and reliable (will work as desired)

1.1.4 Stakeholders

This project aims to be provided as open-source. For this reason, the code published after this work can be used by enterprises who have and want to easily maintain web services in the AWS cloud, like websites, ecommerce or blogs, and make them highly available, secure, robust and fault tolerant.

1.2 Justification

Before starting the project I was wondering if there was a tool or platform that was close or did exactly the same functionality as what I intended to do. However, the closest project I found was LightSail [1], which is an AWS service I will explain in the next section.

Other similar projects could be complex CI/CD pipelines combined with Terraform modules developed by IT departments of enterprises that have services in the cloud. The difference with those and what I want to develop is that those modules and pipelines are built and maintained specifically for that enterprise. I want to make a more generic project that standardizes the deployment of complex web hosting architectures. No messing with the code. No building pipelines. Just enter the specifications of the web hosting you want to create.

1.2.1 Similar Projects

LightSail: It is an AWS service that enables you to quickly launch all the resources you need for small and simple projects, like a WordPress website, a database, storage, etc (as you can see in Figure 1.1). It is straightforward and very easy to use.

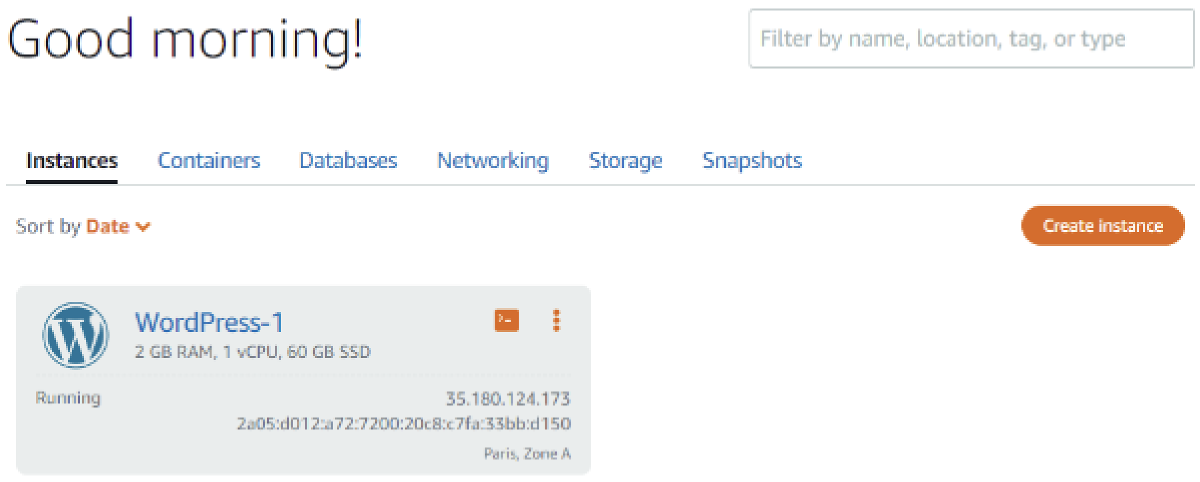


Figure 1.1: LightSail service home page with a wordpress instance running.

In fact, if you want to easily deploy one of the following software of Figure 1.2 and do not get involved in complicated architectural design patters I would choose the LightSail option without any doubt.

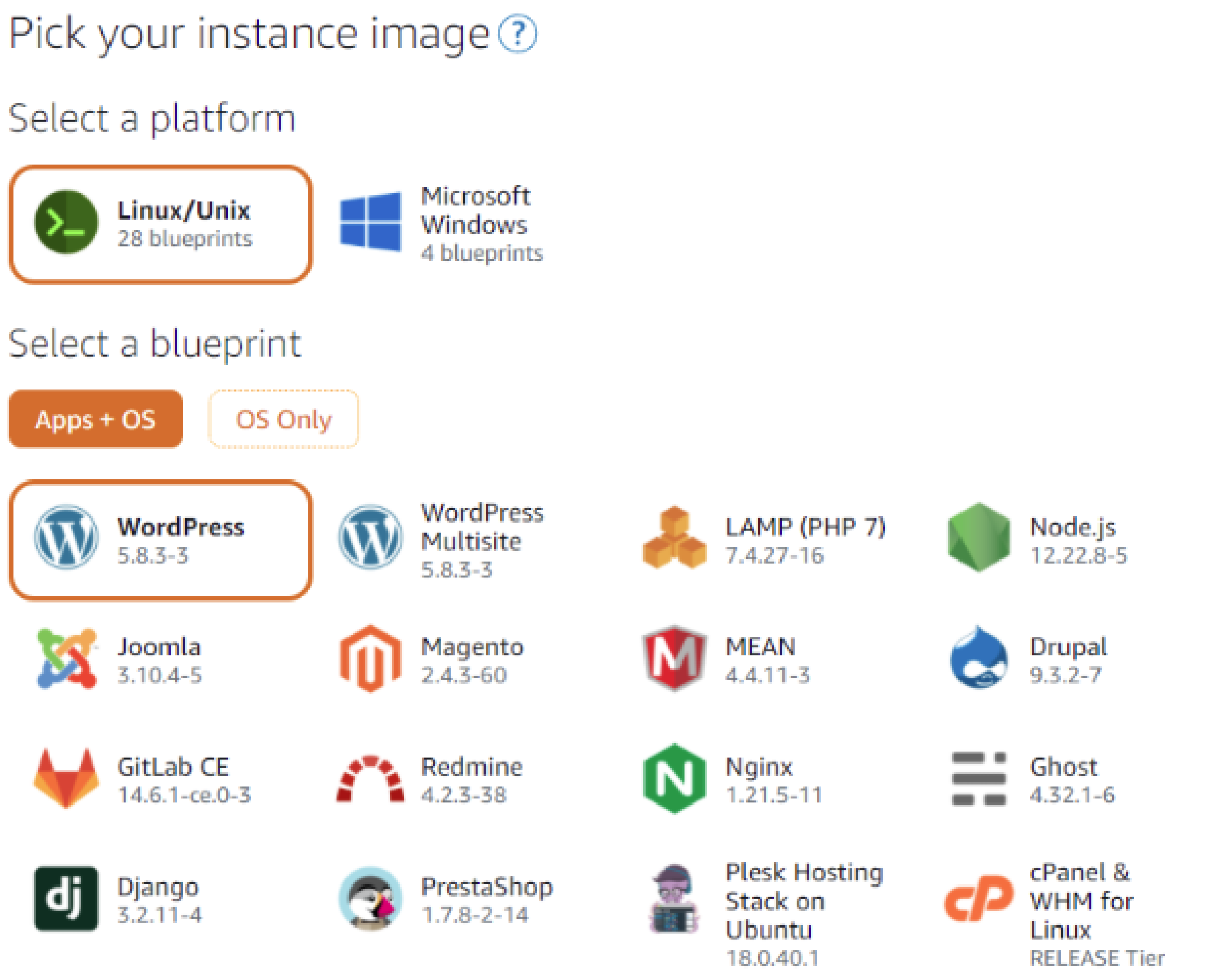
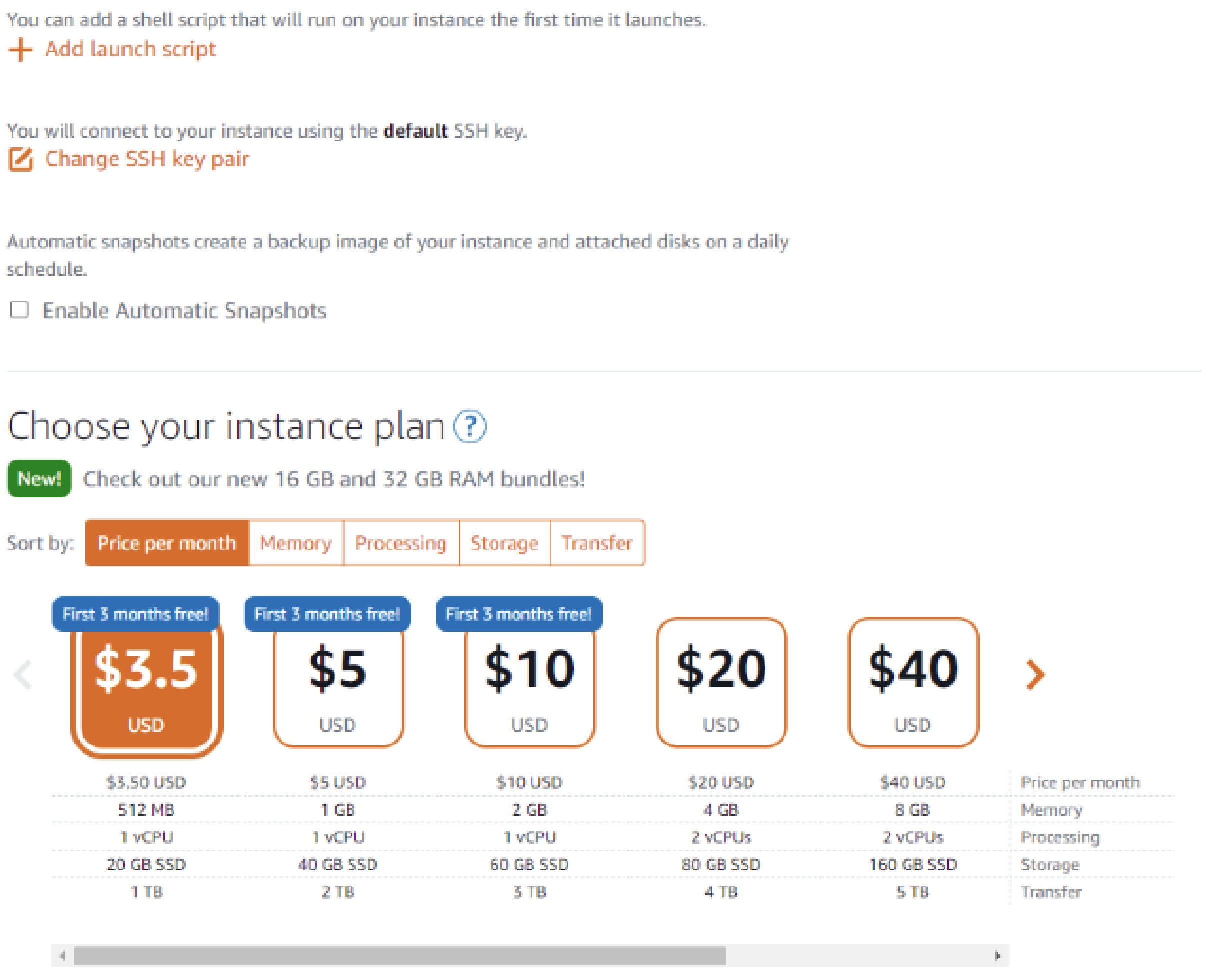


Figure 1.2: Configuring the image of the instance.

You can easily choose pricing options depending on the memory, processing, storage and data transfer you want. In addition you can add a launch script.


You can add a shell script that will run on your instance the first time it launches.

+ Add launch script

You will connect to your instance using the **default** SSH key.

Change SSH key pair

Automatic snapshots create a backup image of your instance and attached disks on a daily schedule.

☐ Enable Automatic Snapshots

Choose your instance plan ?

New! Check out our new 16 GB and 32 GB RAM bundles!

Sort by: Price per month | Memory | Processing | Storage | Transfer

First 3 months free! $3.5 USD	First 3 months free! $5 USD	First 3 months free! $10 USD	$20 USD	$40 USD	
$3.50 USD	$5 USD	$10 USD	$20 USD	$40 USD	Price per month
512 MB	1 GB	2 GB	4 GB	8 GB	Memory
1 vCPU	1 vCPU	1 vCPU	2 vCPUs	2 vCPUs	Processing
20 GB SSD	40 GB SSD	60 GB SSD	80 GB SSD	160 GB SSD	Storage
1 TB	2 TB	3 TB	4 TB	5 TB	Transfer

Figure 1.3: Choosing the instance price plan.

Once the WordPress site is created you can access it via its public IP address and you can destroy it or modify some basic aspects like ip address, firewall or storage at any time.

1.2.2 Justification

The problem with the previous service is that if you want to create more complex web hosting services than can scale horizontally, be high available, use a CDN or be protected with a firewall, it gets way more complicated, because those options are not included in the LightSail service and have to be manually added after. If an enterprise has a critical web portal in the production environment, for sure they want to serve content with less latency and add more instances automatically if there is an increase in inbound traffic.

In addition, with LightSail you do not have the power of the Infrastructure as Code of Terraform, which means the infrastructure is already written and you do not have to create the service every time from the AWS managed console. Furthermore, code can be saved in a cloud repository and is not lost in case someone accidentally deletes the service.

Deploying this kind of infrastructure, even if you count with a cloud architect expert, can take time. With terraform and its Infrastructure as Code potential this process can be automated, only needing to set some variables to actually define the service itself.

1.3 Scope

1.3.1 Objectives and sub-objectives

The main objective of the project is to create a software based tool that makes deploying industry-prepared web hosting to the cloud extremely simple. However this objective is quite general and can be divided into several ones:

- Effectively write clear and robust Terraform files that cover all the important (and not so important) features that all AWS web services should have. Here a list:
  - public IP address
  - Different region availability
  - Different working environment configurations
  - VPC (Virtual Private Cloud)
  - AMI (Amazon Machine Image)
  - subnet
  - security group
  - DNS
  - load balancer
  - target group
  - autoscaling group
  - cloudwatch logs
  - number of instances
  - type of instances (compute power)
  - type of payment (onDemand, spot, etc)
  - roles and permissions
  - web domain

  - storage: S3 service / EBS service
  - databases
  - in-memory cache
  - CDN
  - firewalls
  - NAT gateways

  In addition, I will build those Terraform files with a backend of the Terraform state stored in the S3 service and using state files for each working environment a standard enterprise should have (development and production).

- Create a well architected database using RDS and Aurora service and DynamoDB service for locking the backend.
- Create the front-end of the platform having in mind it has to be simple, secure, direct and easy to use. It is not the main focus of the project. It will be developed using Python.
- Make use of GitHub for version control of the created web services and enhance team based workflows.

There is also another objective that, although is not directly connected to the result of the platform, is really important: during the learning phase of the TFG, I am trying to get the AWS Certified Cloud Practitioner [4] certification and the Terraform Associate Certification [14].

Both will help me better understand those technologies and hopefully will enable me to accomplish all the features I want the platform to have.

1.3.2 Requirements

In order to successfully achieve the completion of the project with the resources and time I have I need to ensure that:

- No more than $50 are spent in AWS during testing the various testing phases (the certification costs are not included).
- The only focus of the project is to create the infrastructure. The software deployed to the infrastructure in order to test it will be extremely basic and/or already created by a third party.

- I will not extend or make any use of LightSail. The platform will be developed from scratch.
- When creating a web service, the given specification must be correct. If you provide the wrong domain name or a wrong type of instance, the web service created may be wrong as there will not be a CI/CD pipeline that tests if the web service is functional.

1.3.3 Potential obstacles and risks

During the TFG planning I have identified the following potential risks:

- Time: The project I want to accomplish is ambitious and if I get stuck in certain parts not all the topics and technologies could be learnt and used successfully, making the project less robust and usable.
- Inexperience with some technologies: Although I have worked as an intern for 5 months using AWS and Terraform I am not proficient at all with them and designing the front end of the platform with a framework I have never used could make the project progress slower.
- Completion of the project in an early stage: I want to dedicate lots of hours to the TFG. If somehow I develop the platform before the expected I can easily extend the project by adding some Ansible playbooks to configure the created infrastructure with a simple application (that could also be stored into a Docker container).
- Bad documentation: One possible risk I need to avoid is to not document everything I do while I am developing the project. It is then easier to write the report of the project and anything is missed.

1.4 Methodology and rigor

1.4.1 Methodology

I will use the Agile methodology. I will manage the project by breaking it into several sub objectives or tasks, also called sprints, and then I will cycle through each one of them planning, executing and evaluating the development of each phase.

Specifically, I will use Trello [24], a web based Kanban project management application that I am familiar with.

Kanban is a Japanese term that means “visual cards”. Hence, this framework used to implement the Agile methodology uses cards or notes distributed in columns, where every card is a task to do. I will divide the word into four columns:

- To do: Including all the tasks that have not been started.
- WIP (Work In Progress): As the name says, this column includes all the tasks that are under development
- Needs review: This column includes all the tasks that have been developed but not tested.
- Completed: Finished tasks.

1.4.2 Validation

I will use GitHub as the version control tool for simplicity and failure recovery. The master branch will include the tested code and every sprint of the Trello board will consist of a new development branch that will not be merged to master until the code is revised and tested.

In addition, I have already scheduled online meetings with the TFG director every three weeks plus a weekly or even daily contact via email when needed with the intention of checking the tasks under development and the project status.

Chapter 2

Time Planning

2.1 Tasks Description

The following tasks will be accomplished from the 22nd of February, the date I started the project, to the end of 27th of June, when I will perform the defense of the bachelor's thesis.

During these 5 months I will dedicate approximately 20 to 25 weekly hours, although it may vary depending on the amount of external work I have. In total, from 450 hours to 550 hours maximum.

In the following sections I will explain how the project is scheduled, taking into account the project management, the learning/formation phase, the project development and the project documentation and its corresponding subtasks. In the 2.1.3 section there is a table summarizing all the tasks.

2.1.1 Task definition

The first phase of the project is the project planning phase and it is composed of the following subtasks:

- T1: TIC tools to support, document and manage the project. In order to accomplish this project successfully I need the best technologies that can allow me to document the project from various devices with synchronized information. I need to have access to the list of tasks defined at any time and have the possibility to organize meetings with the project director telematically.

- T2: Context and scope. I need to indicate the general objectives and the reason why this project is interesting and has viability.
- T3: Time Planning. If I want to accomplish this project in the stipulated deadline I need to carefully plan all the tasks and identify the most important ones. Also plan alternative plans in case there are some troubles.
- T4: Budget and sustainability. When developing a project it is crucial to know its environmental impact that its development will produce. It goes without saying that planning a budget in case there has to be one is of utmost importance.
- T5: Final project definition. I will group all the previous tasks into one final project while I correct the mistakes indicated through the rubrics.
- T6: Meetings and mails with the project director. Online meetings are scheduled every three weeks plus constant communication via mail. We discuss planning, formalities and technology troubles

Overlapping with the project planning, I will be dedicating time to learn about Amazon Web Services and Terraform. I signed up for Cloud Guru, which is a learning platform centered in the cloud and DevOps technologies and will help me get both certifications.

- T7: AWS Certified Cloud Practitioner (CCP). This certification proves the individual who has it has basic and solid concepts of Amazon Web Services and can perform basic to mid-advanced tasks in the AWS environment. The exam fee is $99. The total time of dedication between learning videos, laboratories, study and the exam is approximately 30 hours.
- T8: Terraform Associate Certification. This certification verifies the basic infrastructure automation skills with Terraform that an IT professional should have.The exam fee is $70 and the total time of dedication between learning videos, laboratories, study time and exam is 35 hours.

After this, the development of the project will begin. The development will consist of the following tasks:

- T9: Development of all the Terraform code. Consisting of several terraform files, some of them optional and some of them dependent on others, that replicate the creation of web services.

- T10: Testing of the Terraform files, creation of different environments and remote states in S3 buckets. Once the code is written I will proceed to test it. I will check if all the Terraform files are compatible between them and if the infrastructure created is usable. In addition, I will create 2 different tfstate (terraform state files) files (where the variables will be) to replicate the different working environments of a real company: Development and Production. Finally, I will store the terraform state (the infrastructure as code created) in the cloud as a backup.
- T11: Create the front end of the platform. The front end will be simple and intuitive and will be created with Python. It will enable the user to easily enter the variables to create the web service.
- T12: Create a well architected database in AWS using RDS service and Aurora.
- T13: Create a basic GitHub integration to safely deploy the new code to the cloud. The integration will enhance the team work.

During the development and after it I will be documenting the process (T14), which can also be seen as a task. The last task will be preparing the oral defense (T15), where I will summarize in no more than 30 minutes all the work done and perform a practical demonstration of the platform.

2.1.2 Resources

Human Resources

The human resources are the TFG director Llorenç Cerdà, in charge of mentoring and coordinating during the plan, development and documentation of the project, the GEP tutor Pinto Paola Lorenza, in charge of correcting the project management part and myself, in charge of developing the project.

Material Resources

The project needs some software tools to be accomplished and being well documented plus resources to learn from:

- Learning process
  - A cloud Guru
  - Youtube dedicated channels

- Documentation and meetings
  - Google docs
  - Mendeley
  - Google meet
  - Atenea
  - GanttProject
- Development
  - Visual Studio Code IDE
  - AWS web platform
  - GitHub
  - Python

2.1.3 Summary

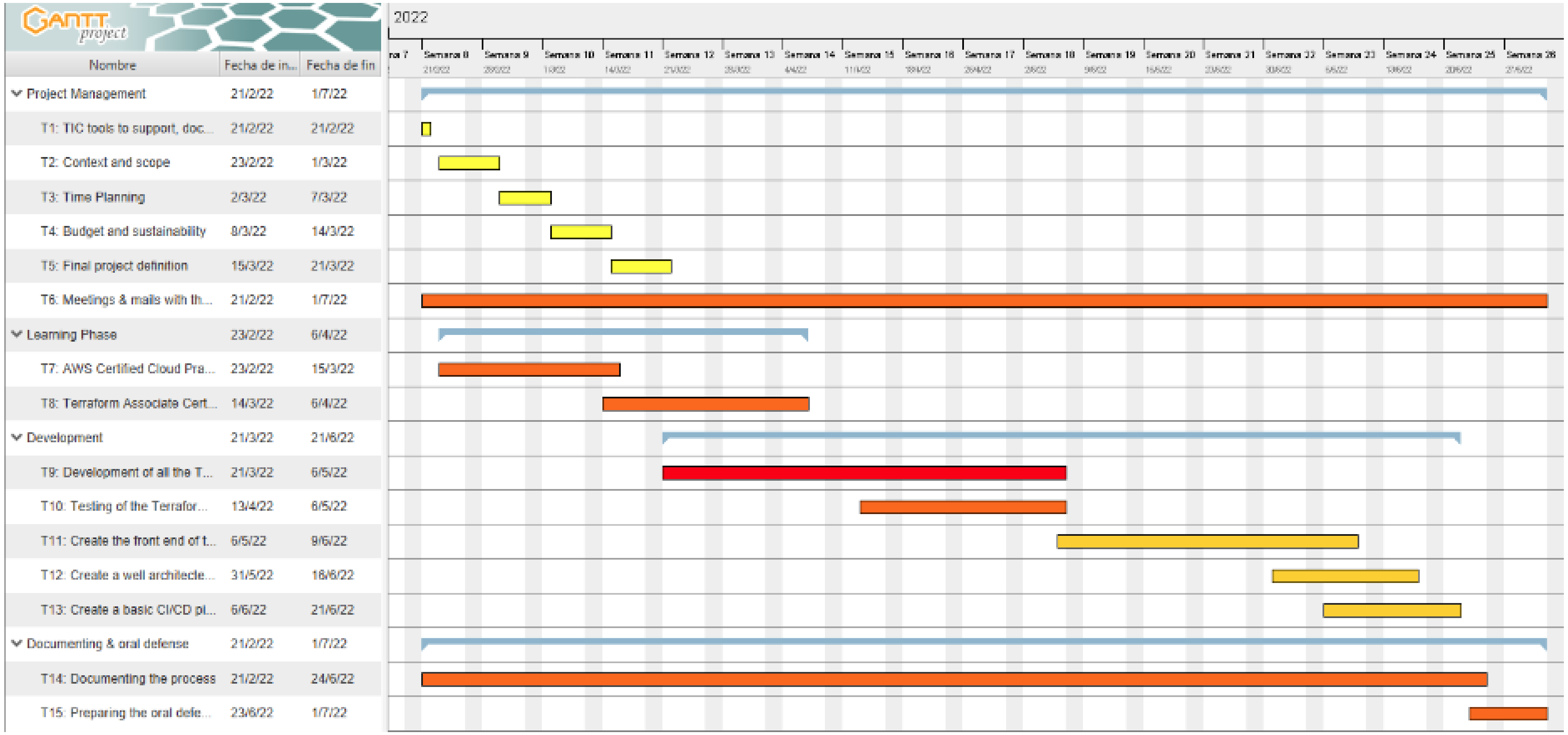
In the following table I summarized the previously explained tasks while showing the dependencies among them and the amount of hours dedicated to each one. The human resource in charge of developing the project is present in every stage of it.

Id.	Task	Time (h)	Dependencies	Specific Resources
T1 - T6	Project Management	80	-	-
T1	TIC tools to support, document and manage the project	5	-	-
T2	Context and scope	20	-	Google Docs, Mendeley, GEP tutor
T3	Time Planning	15	-	Google Docs, GEP tutor. Gantt Project
T4	Budget and sustainability	15	-	Google docs, Mendeley, GEP tutor
T5	Final project definition	10	-	Google docs, Mendeley, GEP tutor
T6	Meetings and mails with the project director	15	-	Google meet, gmail, Atenea, TFG director
T7 - T8	Learning Phase	65	-	-
T7	AWS Certified Cloud Practitioner	30	-	A cloud Guru
T8	Terraform Associate Certification	35	-	A cloud Guru
T9 - T13	Development	230	-	-
T9	Development of all the Terraform code	75	T7,T8	Visual Studio IDE, AWS web platform
T10	Testing of the Terraform modules, creation of different environments and remote states in S3 buckets	35	T9	Visual Studio IDE, AWS web platform
T11	Create the front end of the platform	60	T10	Python
T12	Create a well architected database in AWS using RDS service	30	T9	AWS web platform
T13	Create a basic GitHub integration	30	-	GitHub
T14 - T15	Documenting & oral defense	70	-	-
T14	Documenting the process	60	-	Google docs, Mendeley
T15	Preparing the oral defense	10	T10,T11,T12,T13,T14	-
Total		450		-

Table 2.1: Summary of tasks.

2.2 Estimates and the Gantt

In the next page (Figure 2.1) you can see the Gantt Diagram with an estimation of the completion time of each task. The difficulty of the tasks is shown via a gradient of colors, being yellow the least important task and red the most important task.



Nombre	Fecha de in...	Fecha de fin
Project Management	21/2/22	1/7/22
T1: TIC tools to support, doc...	21/2/22	21/2/22
T2: Context and scope	23/2/22	1/3/22
T3: Time Planning	2/3/22	7/3/22
T4: Budget and sustainability	8/3/22	14/3/22
T5: Final project definition	15/3/22	21/3/22
T6: Meetings & mails with th...	21/2/22	1/7/22
Learning Phase	23/2/22	6/4/22
T7: AWS Certified Cloud Pra...	23/2/22	15/3/22
T8: Terraform Associate Cert...	14/3/22	6/4/22
Development	21/3/22	21/6/22
T9: Development of all the T...	21/3/22	6/5/22
T10: Testing of the Terrafor...	13/4/22	6/5/22
T11: Create the front end of t...	6/5/22	9/6/22
T12: Create a well architecte...	31/5/22	18/6/22
T13: Create a basic CI/CD pi...	6/6/22	21/6/22
Documenting & oral defense	21/2/22	1/7/22
T14: Documenting the process	21/2/22	24/6/22
T15: Preparing the oral defe...	23/6/22	1/7/22

Figure 2.1: Gantt Diagram.

2.3 Risk management: alternative plans and obstacles

During the project, some obstacles and inconveniences may appear. It is important to have this in mind and prepare alternative plans in case.

1. Inability to accomplish the project in the stipulated time. Because of the workload, if I get stuck in some parts this could happen. In this case I would focus on the most important parts of the project: the Terraform files.

   At the same time I would make a really simple user interface locally, do not enter in the GitHub actions implementation and even do not use a database (in this case the infrastructure could not be changed via the user interface).

   As you can see, there are plenty of possibilities to shorten the amount of work. If I choose one or another will depend on the amount of time left.

2. Completion of the project in an early stage. In this case I could extend the project by adding two other technologies:

   - Ansible, which would be really useful configuring the already created infrastructure.
   - Docker, which would help the application by making it less heavy and self contained.

3. Bugs in code or inexperience in Terraform. This case is related to the first one but I wanted to emphasize that if I somehow get stuck with the Terraform part I will extend this task in detriment of creating a better user interface.

   If it happens the other way: if I have some problems completing the user interface, I will not cut any time from other tasks and I will try to make it simpler.

Chapter 3

Budget and Sustainability

3.1 Budget

3.1.1 Identification of costs

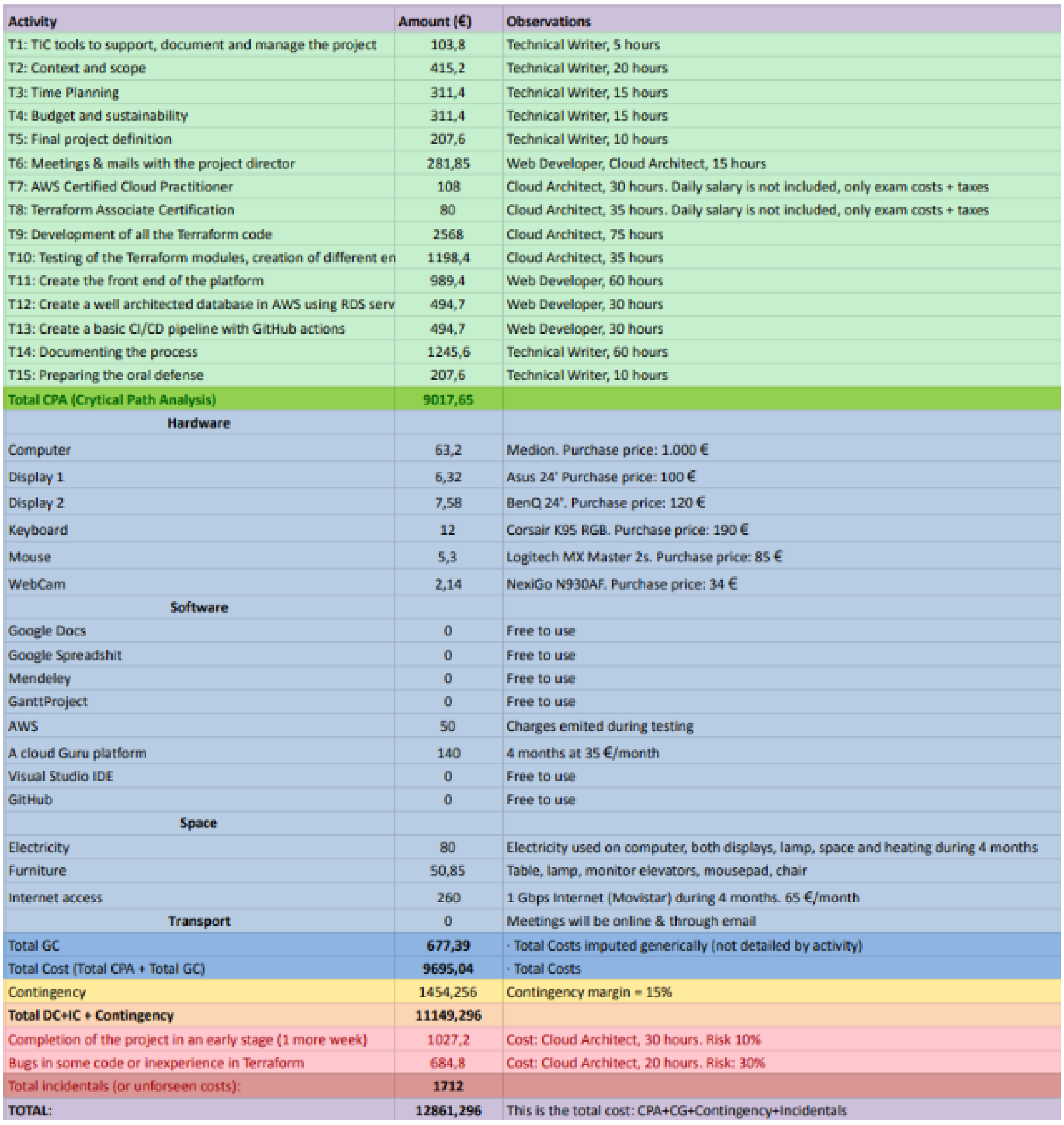
In order to identify all the costs of this project I have to take into account the human resources: the individuals who will execute all the tasks stipulated in the Gantt diagram. In this case, this group of tasks is called CPA (Critical Path Analysis). Notice that testing costs are not included in this chapter and are displayed in section 7.1. To achieve that, I have created three different roles I will assume during the months I will be developing this project. Those are the following:

- Technical writer: A technical writer is the individual in charge of writing and documenting the project. He will be responsible for the tasks T1 to T5, T14 and T15.
- Cloud architect: A cloud architect is the IT professional responsible for overseeing a company's cloud strategy. This includes designing, implementing, managing and monitoring it. In this project he will be in charge of tasks T6 to T10.
- Web developer: A web developer is the professional in charge of creating and maintaining websites and web services. In this project he will be in charge of tasks T11 to T13.

In addition, I will also consider more general costs (GC) such as software and hardware (amortizations), the space in where I am working (which includes electricity costs, furniture and internet access) and the transport fees (which in this case are negligible because I will communicate with my tutor telematically). The space costs are also negligible as we do not pay rent.

Furthermore, in order to be prepared if something goes wrong I also indicate a 15% contingency cost and two different incidents that could also affect the total outcome.

In Table 2.1 I have identified all these resources. For each one I indicate the costs associated and some observations to facilitate the understanding.



Activity	Amount (€)	Observations
T1: TIC tools to support, document and manage the project	103,8	Technical Writer, 5 hours
T2: Context and scope	415,2	Technical Writer, 20 hours
T3: Time Planning	311,4	Technical Writer, 15 hours
T4: Budget and sustainability	311,4	Technical Writer, 15 hours
T5: Final project definition	207,6	Technical Writer, 10 hours
T6: Meetings & mails with the project director	281,85	Web Developer, Cloud Architect, 15 hours
T7: AWS Certified Cloud Practitioner	108	Cloud Architect, 30 hours. Daily salary is not included, only exam costs + taxes
T8: Terraform Associate Certification	80	Cloud Architect, 35 hours. Daily salary is not included, only exam costs + taxes
T9: Development of all the Terraform code	2568	Cloud Architect, 75 hours
T10: Testing of the Terraform modules, creation of different en	1198,4	Cloud Architect, 35 hours
T11: Create the front end of the platform	989,4	Web Developer, 60 hours
T12: Create a well architected database in AWS using RDS serv	494,7	Web Developer, 30 hours
T13: Create a basic CI/CD pipeline with GitHub actions	494,7	Web Developer, 30 hours
T14: Documenting the process	1245,6	Technical Writer, 60 hours
T15: Preparing the oral defense	207,6	Technical Writer, 10 hours
Total CPA (Crytical Path Analysis)	9017,65	
Hardware		
Computer	63,2	Medion. Purchase price: 1.000 €
Display 1	6,32	Asus 24' Purchase price: 100 €
Display 2	7,58	BenQ 24'. Purchase price: 120 €
Keyboard	12	Corsair K95 RGB. Purchase price: 190 €
Mouse	5,3	Logitech MX Master 2s. Purchase price: 85 €
WebCam	2,14	NexiGo N930AF. Purchase price: 34 €
Software		
Google Docs	0	Free to use
Google Spreadshit	0	Free to use
Mendeley	0	Free to use
GanttProject	0	Free to use
AWS	50	Charges emited during testing
A cloud Guru platform	140	4 months at 35 €/month
Visual Studio IDE	0	Free to use
GitHub	0	Free to use
Space		
Electricity	80	Electricity used on computer, both displays, lamp, space and heating during 4 months
Furniture	50,85	Table, lamp, monitor elevators, mousepad, chair
Internet access	260	1 Gbps Internet (Movistar) during 4 months. 65 €/month
Transport	0	Meetings will be online & through email
Total GC	**677,39**	- Total Costs imputed generically (not detailed by activity)
Total Cost (Total CPA + Total GC)	**9695,04**	- Total Costs
Contingency	1454,256	Contingency margin = 15%
Total DC+IC + Contingency	**11149,296**	
Completion of the project in an early stage (1 more week)	1027,2	Cost: Cloud Architect, 30 hours. Risk 10%
Bugs in some code or inexperience in Terraform	684,8	Cost: Cloud Architect, 20 hours. Risk: 30%
Total incidentals (or unforseen costs):	**1712**	
TOTAL:	**12861,296**	This is the total cost: CPA+CG+Contingency+Incidentals

Figure 3.1: Desglosed budget.

3.1.2 Cost estimates

As explained before, the total cost of the project can be estimated by adding the CPA, the GC, the contingency margin and the incident costs.

The CPA is calculated by adding the costs of each task. The cost of one task is equal to the time required for this task multiplied by the hourly pay of the role that is in charge of this task.

In table 2 you can see the gross salaries of each role including social security tax of 35%, the hourly pay and the total role cost for the project. Salaries have been obtained through GlassDoor [13] and the hourly pay is calculated dividing the gross salary plus social security taxes by the 261 working days there are in 2021 and then considering each working day the individual works full-time (8 hours).

For example, a cloud architect hourly pay = 71217.9/(261*8) = 34.10 €/h

The estimated CPA for this project is 9017,65 €

Role	Gross Salary (€)	Gross Salary + SS (€)	Hourly pay (€)	Total role hours	Total role cost (€)
Technical writer	32000	43200	20.68	135	2793,10
Cloud architect	52754	71217.9	34.10	110	3751.9
Web developer	25418	34314.3	16.43	120	1972,1

Table 3.1: Salaries and total costs of each role

In the General Costs (GC), hardware, software and space expenses are included. Hardware costs include:

- Medion Desktop Computer: 1000 €
- Asus Display: 100 €
- BenQ Display: 120 €
- Corsair K95 RGB: 190 €
- Logitech MX Master 2s: 85 €
- NexiGo N930AF webcam: 34 €

To calculate the cost of the hardware material I amortize them by applying the following formula:

amortization = (total cost of the product * project hours) / total hours in 4 years (7120h)

Almost all the software material is free, exceptuating the A cloud guru license in order to obtain the AWS and Terraform certifications (I will pay 35€/month during 4 months) and the AWS expenses resulting from testing the infrastructure I will create. The AWS expenses are set to 50 €, which is the maximum spending I will allow my account to have during these 4 months. The space expenses are spread out between:

- table: 450 €
- chair: 220 €
- lamp: 80 €
- monitor elevators: 20 € x2
- mousepad: 15 €

Space expenses are amortized with the same formula as the hardware material:

amortization = (total cost of the product * project hours) / total hours in 4 years (7120h)

Furniture	Amortized price in €
table 1	28,44
chair	13,90
lamp	5,05
monitor elevators	2,52
mousepad	0,94
TOTAL	50,85

Table 3.2: Furniture amortized expenses

Internet expenses are 65 €/month during 4 months, which is the contract we have at home with Movistar 1Gbps. The electricity cost is an estimation of 20 € per month during 4 months because it is extremely difficult to calculate, especially nowadays.

The contingency costs are equal to the 15% of the CPA+ GC. CPA + GC are equal to 9695,25 € and the 15% of that is 1454,25 €.

Finally, I add the incident costs. Which are the result of possible delays in the project or some bugs in the terraform code, which would require more dedication and thus, more hours of the cloud architect role. The total cost of the project is 12861,29 €.

3.1.3 Management control

Every project must have some sort of procedures to effectively control the defined budget. Of course unnecessary expenses like payment software when there is open-source software available should be avoided. However, there may be some inevitable deviations of the budget and I have to track them. So, for this project I will track the uncontemplated expenses in a spreadsheet with the day, amount and reason of the additional expense. Management control will be performed each time a block (Project management, learning, development, documentation) is finished.

The amount is calculated using the following indicators:

*deviation = (New cost per hour - calculated cost per hour) * total hours of the deviation*

This formula is the same for the GC and Incidents.

To calculate the total deviation we need to add the three indicators:

Total cost deviation = CPA deviation + GC deviation + Incidents deviation

3.2 Sustainability report

The presence of a non-returning point regarding global warming is one of the most important aspects to consider nowadays. The earth has been harmed environmentally during the last centuries and the urge of success and progress prevents humanity from slowing down the disaster. That is why thinking about the footprint of a project is always a must do.

After answering the survey of the EDINSOST2-ODS [10] investigation project I can proudly say I do have some knowledge of the social, economical and environmental aspects an engineering project brings.

Although I lack the metrics and indicators used to measure environmental and social impact I am aware of the best practices and strategies a project should include in order to be environmentally friendly.

During the bachelor degree I have been doing various activities related to this field that made me realize how bad e-waste can be for the environment and that with little effort (like bringing electronic devices to green points, turning off the screens if you leave the work place with the computer open, etc) the life cycle of electronic stuff can be less detrimental to the environment. I normally tend to act accordingly during the completion of all my engineering projects.

Regarding the economic field, I do know how to estimate the economic viability of a project. In fact, during this month we have planned the budget of a serious engineering project and during the bachelor's degree I have attended several subjects (EEE and MI) that have educated me in the economical viability of projects by performing DAFO and CANVAS analysis, cost-benefit charts, etc.

In conclusion, after doing the poll it has made me realize that I was not fully aware of how present and interconnected the three topics (society, economics and environment) were in all the engineering projects and has made me deepen in them a little more.

3.2.1 Economic dimension

Regarding PPP: Reflection on the cost you have estimated for the completion of the project

Taking into account all the human and material resources involved and the benefits this platform could provide I would say the cost is optimum and adjusted. The cost-benefit ratio is acceptable, to say the least.

Regarding Useful Life: How are currently solved economic issues (costs...) related to the problem that you want to address (state of the art)? How will your solution improve economic issues (costs ...) with respect to other existing solutions?

My solution will take benefit of two types of scalability:

- scaling up: incrementing the power of the servers (more CPU, more RAM, etc)
- scaling horizontally: incrementing the number of servers

With these two key aspects the web applications I deploy will adjust to the current demand. If there is almost no traffic the resources will be set to the minimum, however if, for example, the CPU usage is greater than 80%, my code automatically deploys another instance.

With this I always ensure the minimum costs, and this can only be done if you work with big cloud providers, like AWS. It is impossible with an on-premise approach.

In addition my solution benefits from massive economies of scale, like AWS: millions of customers make cheaper prices and discounts possible.

3.2.2 Environmental dimension

Regarding PPP: Have you estimated the environmental impact of the project?

Although there are lots of factors and aspect that could directly or indirectly affect the environment I consider that:

- Working from home
- Contacting with the tutor via online meetings and mailing
- Developing the project in a cloud provider with its servers being powered by green energy

Means that this project will have significantly less environmental impact than others that imply transport and throwable hardware and as far as I know will be determined by the personal hardware I use and the servers in the cloud I use during the development and testing phases.

Regarding PPP: Did you plan to minimize its impact, for example, by reusing resources?

As far as physical resources, I will be using the same hardware the whole project: my personal computer, keyboard and mouse. However, I cannot decide which servers I am using in the cloud as AWS is in charge of this.

Regarding Useful Life: How is currently solved the problem that you want to address (state of the art) How will your solution improve the environment with respect to other existing solutions?

Other similar solutions may run in servers powered by brown energy. My case ensures that 100% of the energy is green, as stated in the official AWS documentation, all of its data centers run with renewable energy.

3.2.3 Social dimension

Regarding PPP: What do you think you will achieve -in terms of personal growth- from doing this project?

Firstly I will gain a lot of experience for my professional career, as I want to work as a cloud architect. All the technologies involved in the creation of the platform will be used by me in the future. Secondly, by the end of this project I will be capable of managing important full-stack projects that involve different technologies that need to fit with each other. Finally, I will learn to document more efficiently and professionally.

Regarding Useful Life: How is currently solved the problem that you want to address (state of the art)?

This problem does not have a real solution as far as I know. Currently, the creation of web services in the cloud is managed by IT professionals with knowledge of cloud providers and / or infrastructure as code.

The closest solution I know is LightSail, an AWS service that aims to solve the creation of web service but has little to no control over the infrastructure created and is not compatible with IaC, although for creating extremely simple WordPress websites there is no better and time-efficient solution than this one.

Regarding Useful Life: How will your solution improve the quality of life (social dimension) with respect to other existing solutions? Is there a real need for the project?

Although this project does not have a direct impact in the improvement of the quality of life, if developed successfully, it would simplify lots of tedious daily processes. Doing things in less time is an improvement.

Yes, this project aims to enable the creation of simple to mid complex web services in the cloud by non-educated people thanks to its friendly user interface and multiple automated processes. Thus, it could be used by real companies and individuals on a daily basis.

Chapter 4

Terraform

Terraform is an open source infrastructure as code software tool that uses HashiCorp Configuration Language (HCL), which is JSON-like, and lets DevOps teams configure cloud and on-premises resources in a human-readable configuration that can be versioned and reused. It basically lets engineers manage, monitor and provision resources through code instead of manually in person. Terraform files have a .tf extension.

4.1 HCL Syntax

HCL is a declarative language code. In the official documentation is defined as ”Describing an intended goal rather than the steps to reach that goal” [23] which means that instead of having to program the program logic, the user tells terraform the result he wants to obtain and lets Terraform figure out the necessary steps to do so.

In order to resolve this steps, Terraform creates a dependency tree of all the resources. This is really important because in cloud environments, resources have an order of creation. For example, in AWS you can not create a subnet if you do not have a VPC. The terraform graph command shows this dependency graph. Unfortunately I can not show the one of my project because it is too big (there are huhndreds of dependencies). Instead, I show a really simple graph of a google cloud deployment.

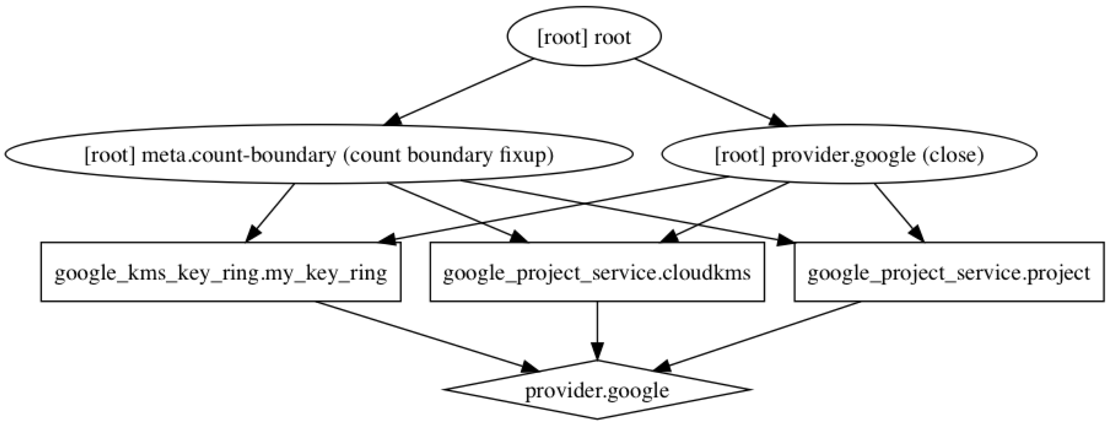
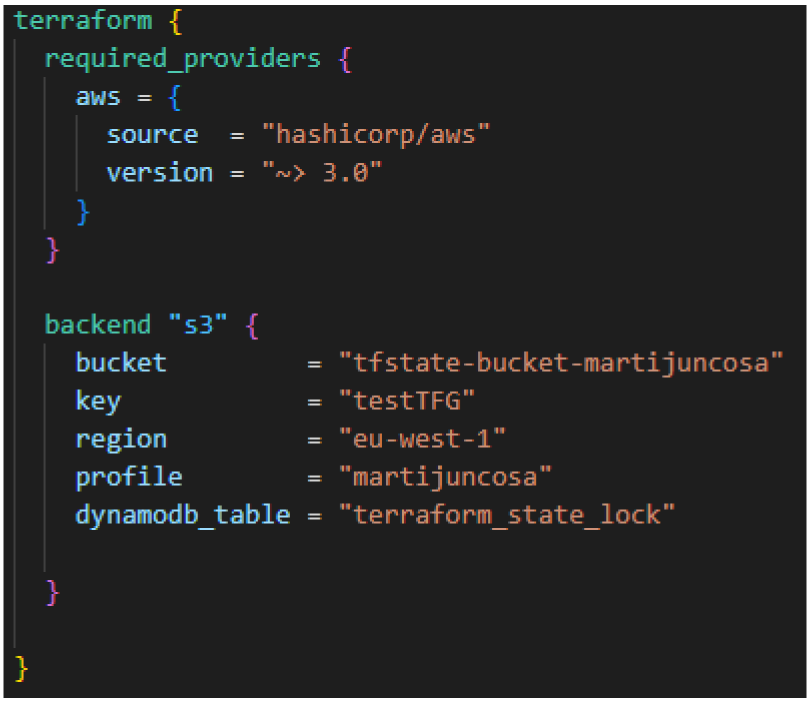

[root] root
[root] meta.count-boundary (count boundary fixup)
[root] provider.google (close)
google_kms_key_ring.my_key_ring
google_project_service.cloudkms
google_project_service.project
provider.google

Figure 4.1: Dependency graph. Source: gihub.com

And how does HCL lets the user tell what to do? With blocks

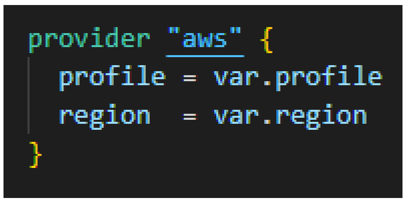
- Terraform block: The principal block of the configuration. It is mainly used to define the required provider and the backend (which I will explain in a few sections).



```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }

  backend "s3" {
    bucket         = "tfstate-bucket-martijuncosa"
    key            = "testTFG"
    region         = "eu-west-1"
    profile        = "martijuncosa"
    dynamodb_table = "terraform_state_lock"

  }

}
```

Figure 4.2: Terraform block

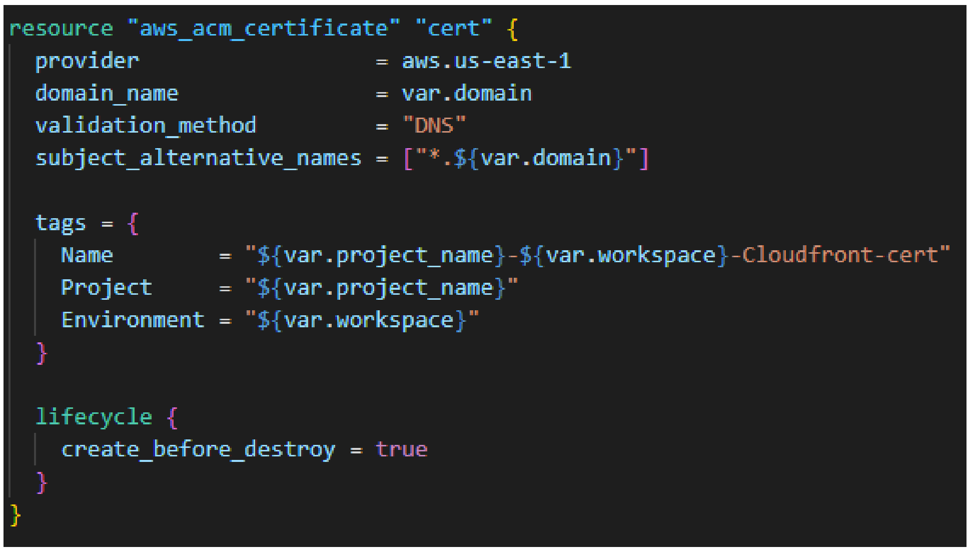
- Provider block: Once the required provider is specified, a provider block must be used to set the provider local name (the provider name that will be used in the configuration) along with the credentials to access it and the region in which it will operate.



```
provider "aws" {
  profile = var.profile
  region  = var.region
}
```

Figure 4.3: Provider block

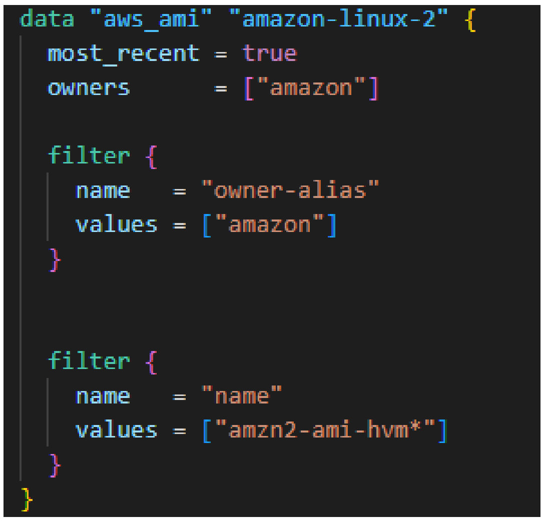
- Resource block: It is the most important element in the Terraform language. Each resource block is used to describe an infrastructure element of the given provider.



```
resource "aws_acm_certificate" "cert" {
  provider                  = aws.us-east-1
  domain_name               = var.domain
  validation_method         = "DNS"
  subject_alternative_names = ["*.${var.domain}"]

  tags = {
    Name        = "${var.project_name}-${var.workspace}-Cloudfront-cert"
    Project     = "${var.project_name}"
    Environment = "${var.workspace}"
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

Figure 4.4: Resource block

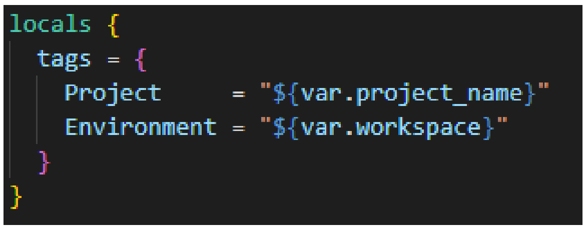
- Data block: This block is used to retrieve information of already created resources for the Terraform configuration.



```
data "aws_ami" "amazon-linux-2" {
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "owner-alias"
    values = ["amazon"]
  }


  filter {
    name   = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
```

Figure 4.5: Data block

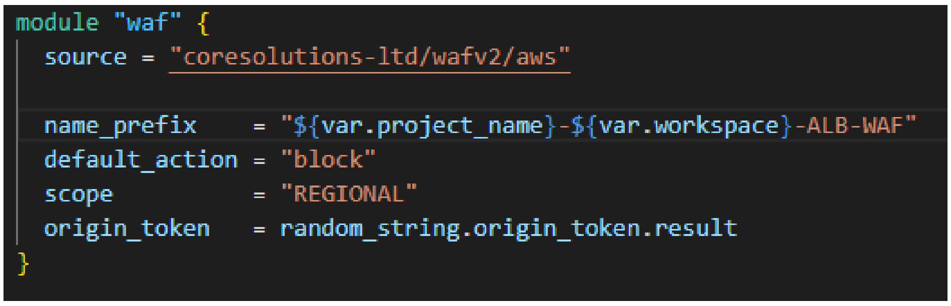
- Locals block: This block permits the definition of local values. Local values can be seen as local variables and can be a constant expression or equal to a variable.



```
locals {
  tags = {
    Project     = "${var.project_name}"
    Environment = "${var.workspace}"
  }
}
```

Figure 4.6: Locals block

- Module block: Modules are defined with module blocks. A module is a container of multiple terraform files that provide a certain functionality and can be used in your current configuration. In my project I use two modules, one for creating an application firewall and one for creating a complex database.



```
module "waf" {
  source = "coresolutions-ltd/wafv2/aws"

  name_prefix    = "${var.project_name}-${var.workspace}-ALB-WAF"
  default_action = "block"
  scope          = "REGIONAL"
  origin_token   = random_string.origin_token.result
}
```

Figure 4.7: Module block

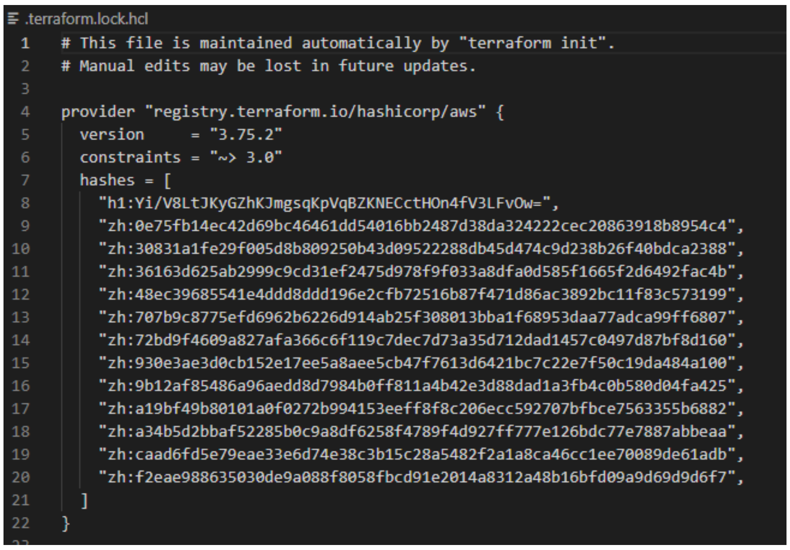
Terraform is flexible with indentation, but has some conventions. terraform fmt command automatically applies them and rewrites misconfigured files with a canonycal style, as can be seen in section 6.7

4.2 Providers

In order to work with different clouds (AWS, Azure, Alibaba Cloud, etc) Terraform uses plugins called providers. Each provider is maintained and distributed separately from Terraform itself and has its release and version numbers. Each provider defines a series of resources and data blocks and without them Terraform cannot manage any kind of infrastructure. Even private providers can be created for managing private resources, but that goes beyond the scope of this project.

To use a provider, it just has to be defined in your configuration inside a required providers clause, with its source and version. Then a provider block can be used to define the provider local name.

Once the provider is set and the configuration is initialized, terraform automatically creates the terraform dependency lock file. This file ensures that external dependencies such as providers and modules are correctly tracked and do not generate dependency issues if changed.



```
.terraform.lock.hcl
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/aws" {
  version     = "3.75.2"
  constraints = "~> 3.0"
  hashes = [
    "h1:Yi/V8LtJKyGZhKJmgsqKpVqBZKNECctHOn4fV3LFvOw=",
    "zh:0e75fb14ec42d69bc46461dd54016bb2487d38da324222cec20863918b8954c4",
    "zh:30831a1fe29f005d8b809250b43d09522288db45d474c9d238b26f40bdca2388",
    "zh:36163d625ab2999c9cd31ef2475d978f9f033a8dfa0d585f1665f2d6492fac4b",
    "zh:48ec39685541e4ddd8ddd196e2cfb72516b87f471d86ac3892bc11f83c573199",
    "zh:707b9c8775efd6962b6226d914ab25f308013bba1f68953daa77adca99ff6807",
    "zh:72bd9f4609a827afa366c6f119c7dec7d73a35d712dad1457c0497d87bf8d160",
    "zh:930e3ae3d0cb152e17ee5a8aee5cb47f7613d6421bc7c22e7f50c19da484a100",
    "zh:9b12af85486a96aedd8d7984b0ff811a4b42e3d88dad1a3fb4c0b580d04fa425",
    "zh:a19bf49b80101a0f0272b994153eeff8f8c206ecc592707bfbce7563355b6882",
    "zh:a34b5d2bbaf52285b0c9a8df6258f4789f4d927ff777e126bdc77e7887abbeaa",
    "zh:caad6fd5e79eae33e6d74e38c3b15c28a5482f2a1a8ca46cc1ee70089de61adb",
    "zh:f2eae988635030de9a088f8058fbcd91e2014a8312a48b16bfd09a9d69d9d6f7",
  ]
}
```

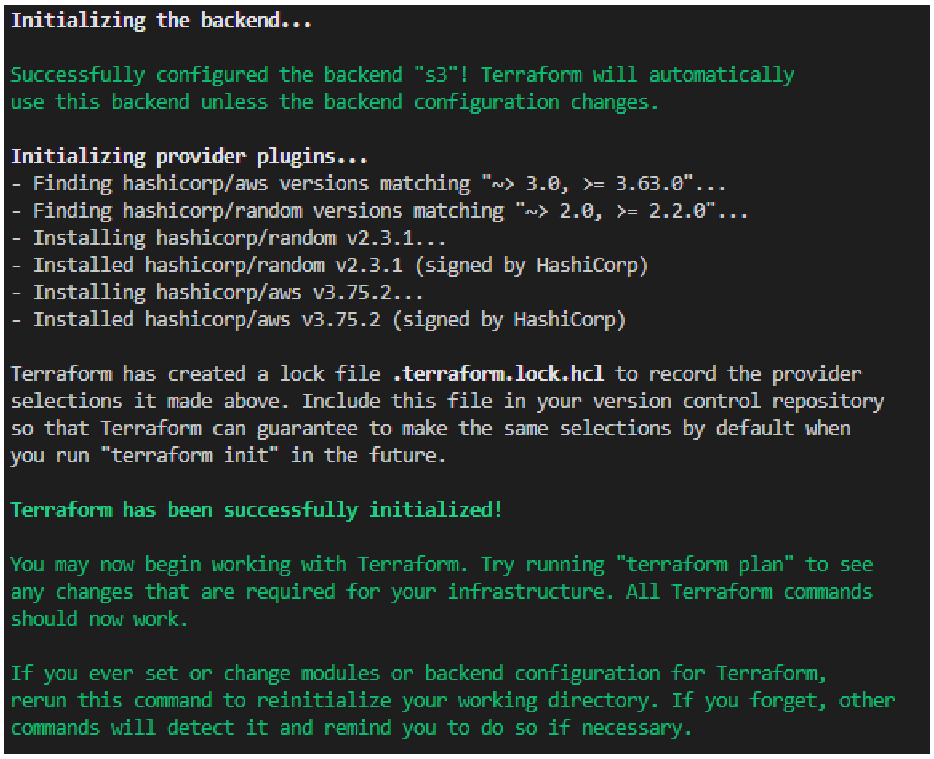
Figure 4.8: Terraform dependency lock file

4.3 Stages (core commands)

Once the terraform configuration is written, there are some steps to follow before deploying the infrastructure.

- terraform init: This command initializes the current directory. This is the first command that should be run. It initializes the backend, the provider plugins and updates the workspace. If using the -upgrade option, it upgrades the modules, searching for new releases.

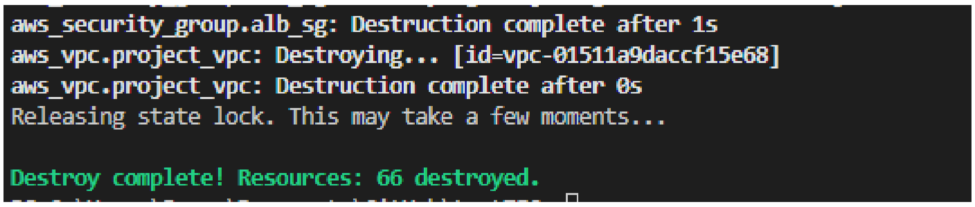
  Then, it stores all this metainformation under the .terraform folder. Finally, it creates the terraform.lock.hcl (the dependency lock file). It can be run safely multiple times.



```
Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.

Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 3.0, >= 3.63.0"...
- Finding hashicorp/random versions matching "~> 2.0, >= 2.2.0"...
- Installing hashicorp/random v2.3.1...
- Installed hashicorp/random v2.3.1 (signed by HashiCorp)
- Installing hashicorp/aws v3.75.2...
- Installed hashicorp/aws v3.75.2 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Figure 4.9: terraform init -upgrade command

- terraform plan: This command is really useful because it lets the user preview the incremental changes that Terraform has calculated for making your infrastructure. It does not deploy anything and is really safe and recommended to use before applying the configuration. It reads the current state file and based on that outputs the resources that will be added, destroyed or changed.
- terraform apply: The apply command performs the same steps as the plan command, however after prompting all the calculated changes, it asks the user if he really wants to apply the configuration. If used with the -auto-approve the command directly applies the infrastructure without asking the user, which is really usable for automating purposes.
- terraform destroy: Essentialy it is a terraform apply with the completely opposite purpose. When the terraform state file has resources tracked this command can be used to safely delete all the deployed infrastructure taking into account all the dependencies.



```
aws_security_group.alb_sg: Destruction complete after 1s
aws_vpc.project_vpc: Destroying... [id=vpc-01511a9daccf15e68]
aws_vpc.project_vpc: Destruction complete after 0s
Releasing state lock. This may take a few moments...

Destroy complete! Resources: 66 destroyed.
```

Figure 4.10: terraform init -upgrade command

4.4 State File

The terraform state file is a file written in JSON format used to store data about the managed infrastructure by mapping real world resources that have been deployed to your configuration.

Thanks to this file, Terraform can keep track of all the metadata and given the case of a new terraform apply with new code, a terraform destroy or a completely new infrastructure set up, Terraform knows which incremental changes has to make.

If the file is saved locally it is named terraform.tfstate. However, the normal case for it is to be stored remotely, as explained in the next section. In that case, the user defines its name.

4.5 Backend

Backends define where Terraform's state files are stored. By default, the backend is called local and stores the state file as a local file in the disk. However, depending on the provider used it may be helpful to store the state remotely.

In my project I use the AWS provider, which allows the user to define an AWS S3 bucket (a object storage unit) as the remote backend. In addition, AWS supports locking the state while operations are being performed thanks to a service called DynamoDB, which is a NoSQL database. Its setup is really easy, the only important thing is that the partition key has to be named LockID and be string based.

Using a backend is specially helpful when working with collaborative environments because it helps prevent conflicts. Two people cannot update the state file at the same time.

4.6 Workspaces

Workspaces enable the user to have more than one state file associated with the same configuration. Initially there is only one workspace, called default. However, depending on the used backend, a user can create other workspaces. Local and AWS S3 backends support this feature.

This feature permits DevOps teams to test the created infrastructure in a dev workspace and then deploy the production workspace when everything is working.

Backends are created with: terraform workspace new workspace_name And are selected to be used with: terraform workspace select workspace_name

Chapter 5

Configuration of the HA, secure and scalable infrastructure

5.1 AWS global infrastructure

Before starting explaining the different components of the project, I find important to introduce the AWS global infrastructure [5] among some core concepts that will be constantly repeated during the following sections.

AWS certainly reaches more parts of the world that any other public cloud provider in the world. That is undoubtedly one of the reasons why they have more than the 30% of the cloud market share. AWS global infrastructure components have a specific terminology and each one is key to understand how their hundreds of services work.

- AWS Regions: a region is a physical location around the world. Each region holds multiple availability zones, which are physically separated and isolated from one another. Each region meets the highest levels of security, data protection and compliance. Currently there are 26 launched regions and 8 more to come. Most regions have 3 or more AZs.
- AWS Avaliability Zones (AZs): one or more data centers with redundant power, connectivity and networking in the same AWS region. All AZs in the same region are interconnected with high-bandwith and ultra-low-latency networking, enough to provide synchronous replication between AZs. Users can make use of those different AZs to build applications which are more fault tolerant, highly available and scalable. In total

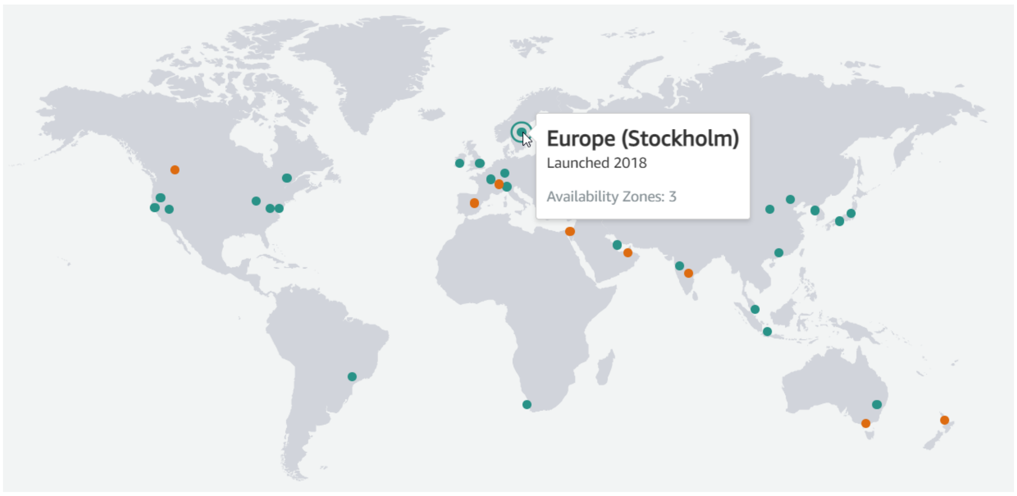


Figure 5.1: Map of AWS regions and its AZs. Source: aws.amazon.com

there are 84 deployed AZs.

- AWS Edge Locations: Edge locations are endpoints of the AWS global infrastructure used for caching content as part of the AWS CDN, Amazon CloudFront. There are more than 400 edge locations deployed.
- AWS Local Zones: Local zones are parts of the AWS infrastructure used to run latency sensible applications like streaming, real time gaming, augmented reality, etc. Essentially, compute, storage, database and other few AWS services can be placed closer to the audience. They are not part of an AZ nor a region. Like edge locations, they are counted apart. Currently there are 17 local zones deployed.
- AWS wavelength zones: wavelength zones enable developers to build applications that provide single-digit millisecond latencies to mobile phone users. Currently there are 28 of them.

Thanks to that infrastructure, close to 250 countries are effectively served with a high availability, low-latency, fault tolerant and physically secure infrastructure that sets AWS appart in the cloud panorama.

5.2 Defining the deployment

The aim of the platform I have built is to deploy the architecture below (Figure 5.2). The ultimate goal of this infrastructure is to serve as a highly available, fault tolerant and secure web hosting, specially designed to accommodate a WordPress installation, although any type of CMS or private software could be used.

This deployment benefits from the AWS Public Cloud infrastructure and the wide variety of services that AWS offers in order to offer several options in some of its components, making it even more configurable. And thanks to the automation capabilities of Terraform can be deployed in much more less time.

I got inspired by an Amazon white papers [26] and some personal projects [9, 28]

In this section I am going to discuss and explain each and every component of this infrastructure, which has been built in a secure, cost-effective and efficient way.

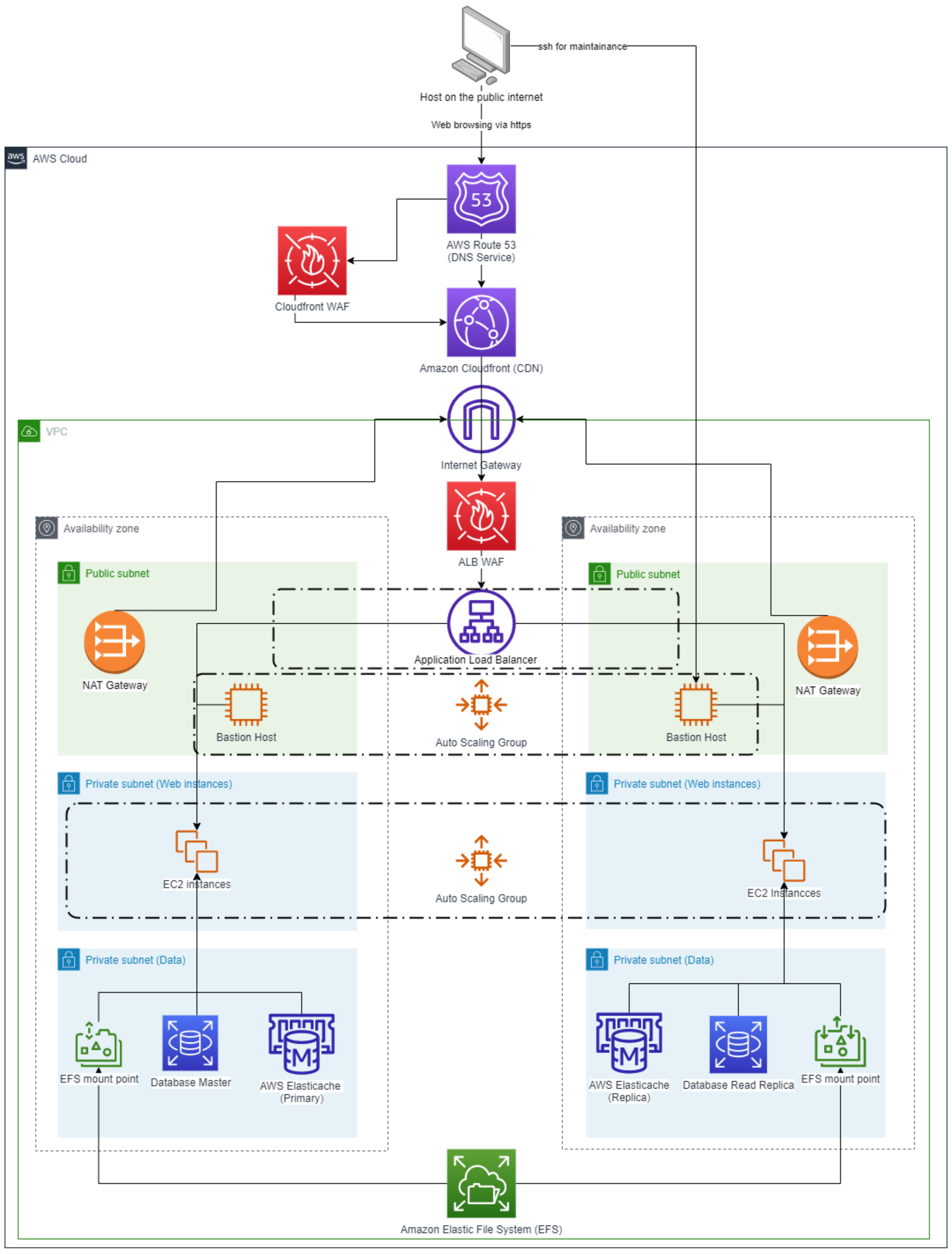


Figure 5.2: Infrastructure to be deployed

Note that route tables, security groups, SSL certificates, target groups and other complementary AWS services that actively contribute to the correct functioning of this infrastructure are not displayed in this diagram in order to make it more understandable. However, all of them are explained in detail during this section.

5.3 Networking

Networking is (alongside EC2) the most used cluster of services in AWS. It serves as the foundation of most AWS projects and a good design an implementation can make a difference in the security and organizational aspects.

5.3.1 VPC

A VPC or Virtual Private Cloud is a logically isolated slice of cloud where resources can be launched into. You can define multiple VPCs in your AWS account. Inside a VPC you gain full control over the networking environment. It is basically the base in which you will build your infrastructure.

In order to create a VPC a CIDR block has to be defined. I use 10.0.0.0/20, which guarantees 2^{12} hosts.

A VPC is located in one region (although cross-region VPC can be created using VPC peering). In my project, the region can be specified via variable.

5.3.2 Subnets

Subnets are logical partitions of the VPC IP address pool, the CIDR block designated to each subnet of the same VPC must be different and the netmask number must be larger than the one defined in the VPC. In my project I am using /24 subnet netmasks which provide 256 theoretical hosts available per subnet (in practice, only 251 are available due to reserved IPs for routing purposes). Inside a subnet the user can launch AWS resources like EC2 instances.

Automatic allocation of public IPv4 addresses to the instances created inside a subnet can be specified. If the aim is to create a public subnet this option must be enabled while if it is a private subnet this must be disabled.

In my project I am defining 6 subnets: 2 private subnets designated to storage (databases, a common storage point, called EFS, and the database in-memory cache). 2 private subnets dedicated to the servers (instances) and two public subnets dedicated to the NAT gateways of the private subnets, to the bastion hosts and, of course, with access to an internet gateway which offers access to the public internet.

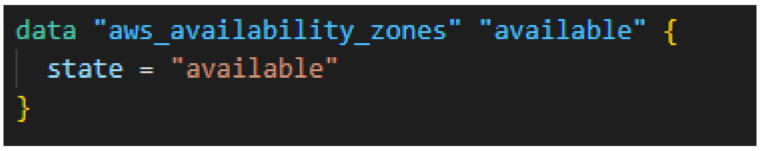
Private subnets offer an implicit layer of security to the resources deployed inside. The servers that will run the software and databases need to run there because there is no direct

Subnets	CIDR block	Public IP option
Public Subnet 1	10.0.0.0/24	Enabled
Public Subnet 2	10.0.1.0/24	Enabled
Private Web Subnet 1	10.0.2.0/24	Disabled
Private Web Subnet 2	10.0.3.0/24	Disabled
Private Data Subnet 1	10.0.4.0/24	Disabled
Private Data Subnet 2	10.0.5.0/24	Disabled

Table 5.1: Subnet summary

access from the public internet (they are not provided a public IP address), instead, public users that want access to this resources will have to enter from another point, like a load balancer protected by a firewall.

Notice the fact that I define two subnets of each type. This is because of the High Availability design pattern: at the end of the day all the infrastructure will be replicated in two different availability zones of the chosen region.



```
data "aws_availability_zones" "available" {
  state = "available"
}
```

Figure 5.3: Data block

In my terraform configuration, I obtain the available AZs via a data block. This block searches for the availability zones that are working in the region that has been specified in the VPC and handles the result as an array with the names of the AZs.

5.3.3 Security Groups

Security groups are virtual firewalls that control the traffic that is allowed to reach (inbound traffic) and leave (outbound traffic) the resources that are associated with. This is done by adding inbound and outbound rules specifying the source (an IP address, a set of IPs, all the IPs, the resources that have a certain security group associated, etc), a port and a protocol. This rules are only allowing, there are not deny rules in security groups. Everything that is not allowed is implicitly denied.

They, however, cannot prevent SQL injection attacks nor cross site scripting (for that you need to use WAF - Web Application Firewalls).

In my project I define 7 security groups for the different components that will be launched, specifying certain ports and sources.

Security Group	Ports allowed	Source
bastion-sg	22	personal IP
web-public-sg	80, 443, 22	alb-sg (80,443), 0.0.0.0/0 (22)
web-private-sg	80, 443, 22	alb-sg (80,443), web-public-sg (22)
alb-sg	80, 443	0.0.0.0/0
rds-sg	3306	web-private-sg
efs-sg	2049	web-private-sg
elasticache-sg	11211 (memcached) or 6379 (redis)	web-private-sg

Table 5.2: Security Groups summary

Finally, another remarkable aspect about security groups is that they are stateful. This means that if an instance recieves allowed inbound traffic, it is allowed to leave regardless of the outbound rules, and at the same time, if an instance sends a request, the response is allowed despite the inbound rules set.

This is important because another similar resource called network ACL is stateless, meaning that return traffic must be explicitly allowed by rules. ACLs provide an additional layer of defense if security groups are too permissive because they support allow and deny rules. Network ACLs have not been used in this project because security groups are strict enough.

5.3.4 Internet Gateway

The internet gateway, as the name suggests, is the gateway point of the VPC to the public internet. Even if an instance had a public ip address associated but happened to be located in a VPC without an Internet Gateway, it would not have connectivity. It is redundant and highly available and does not limit the bandwith of internet connectivity.

It basically serves two purposes: providing a target for the route tables in case they need to route internet traffic and perform NAT for instances that have a public IPv4 address.

Even if they have a public IPv4 address, instances are only aware of its internal ipv4 address. That is why IGW gets in charge of this and provides one-to-one NAT on behalf of the instance. Only one IGW can be assigned to a VPC.

5.3.5 NAT Gateway

Instances in the private subnets need to have access to the public internet in order to download software for patching. In this infrastructure configuration instances are located in a private subnet, and thus, none of them has a public IP address. Translation between private and public ip addresses is needed, this is where NAT gateway comes into place.

It is quite similar to the IGW, but the main difference resides in that it works for private instances and it only works one way, meaning that it cannot be reached from the internet unless explicitly allowed.

I want a high available and fault tolerant NAT gateway deployment. If the user has resources in multiple Availability Zones and they share one NAT gateway, once the NAT gateway's Availability Zone is down, resources in the other Availability Zones will lose internet access. To avoid that and create an Availability Zone-independent architecture, I create a NAT gateway in each Availability Zone and configure the routing to ensure that resources use the NAT gateway in the same Availability Zone they are located.

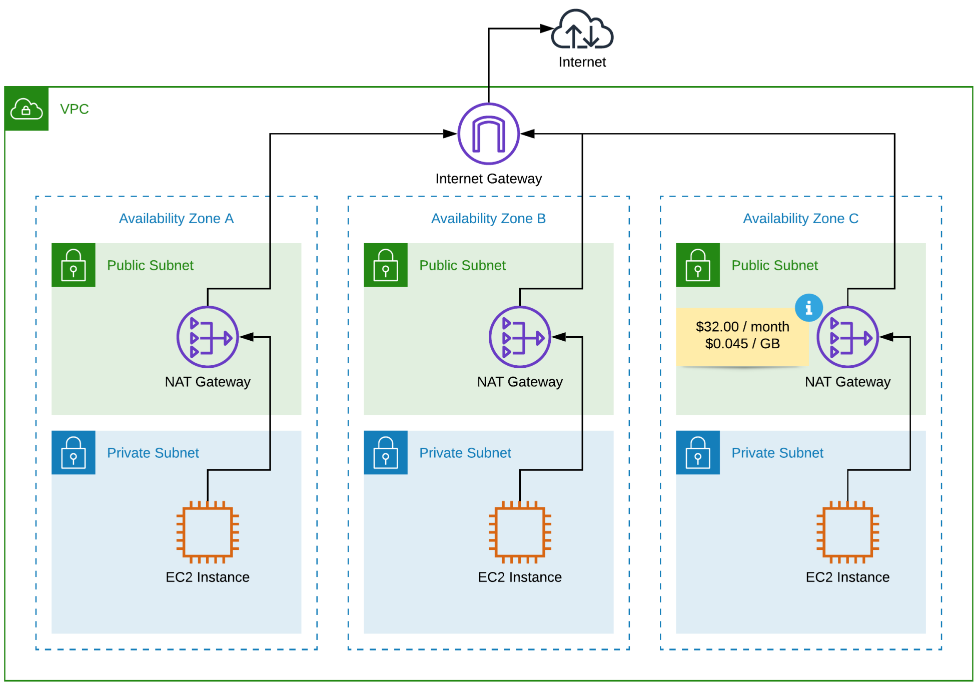


Figure 5.4: Multiple NAT Gateways. Source: stackoverflow.com

In addition, each NAT gateway is assigned an EIP (Elastic IP Address). An EIP is an unchangeable Ipv4 address that Amazon itself delegates from its IPv4 public block of addresses. In case the NAT gateway is destroyed and recreated the IP will remain the same and route tables will not have to be modified.

5.3.6 Route Table

I create one public route table where I create a rule that indicates that all traffic of the public subnets of both availability zones must go to the internet gateway and two private route tables, one for each NAT gateway, where I indicate that all the egress traffic of the private subnets dedicated to the instances goes into the specified NAT gateway.

With this implementation all the resources deployed in this networking configuration except databases and in-memory caches, which are AWS managed services and do not need patching from the user side, will have the possibility to initiate (and sometimes receive) a request to the public internet or be completely isolated from it.

5.4 Load Balancing

Load balancers distribute inbound traffic across multiple targets in one or more AZs to reduce the load that each one receives. They help to deliver applications that are high available and scale in and out automatically and can monitor its health and performance. In AWS there are 3 types of load balancers:

- Classic Load balancer: Intended for the old classic EC2 instances. Not recommended.
- Network Load balancer: Works in the networking layer of the OSI model. Extremely good performance, especially with millions of requests per second, scales impressively and is the most expensive option.
- Application Load balancer: ALB (Application Load Balancer): Works in the application layer of the OSI model and thus can inspect all the packets that receives.

I have chosen to use ALB because it has the functionality to distinguish traffic for different targets (mysite.co/accounts vs. mysite.co/sales vs. mysite.co/support) and distribute traffic based on rules for target group, condition, and priority. In each ALB there are two components that are worth to be mentioned:

- target group: In each target group you can register targets (EC2 instances). Each target group is used to route requests to those registered targets, in my case, from an ALB. In addition, health checks can be enabled.

  Thanks to them, the ALB can identify if a target is healthy or unhealthy and stop routing traffic to them, which otherwise would be lost.

- listener: At least one listener is required in a load balancer. A listener receives connections in the specified port and with the specified protocol. For example 443 - https.

  Then, based on the defined rules, it forwards the request to its registered target groups. Listeners can also have an SSL certificate embedded and perform SSL offloading, which consists on decrypting the SSL traffic.

In my project, I use an internet facing ALB that spans across both public subnets and has a listener in the 443 port that receives encrypted HTTPS traffic from the CDN, as I will explain later. Then thanks to the SSL certificate it performs SSL offloading and forwards the request to the unique target group of the configuration.

As the ALB spans across both AZs, instances from both AZs can be registered to the target group. Registered instances will receive the traffic in a round-robin way and its health status will be checked each 10 seconds in the port 80 using HTTP protocol with the specified path (index.php, index.html, etc)

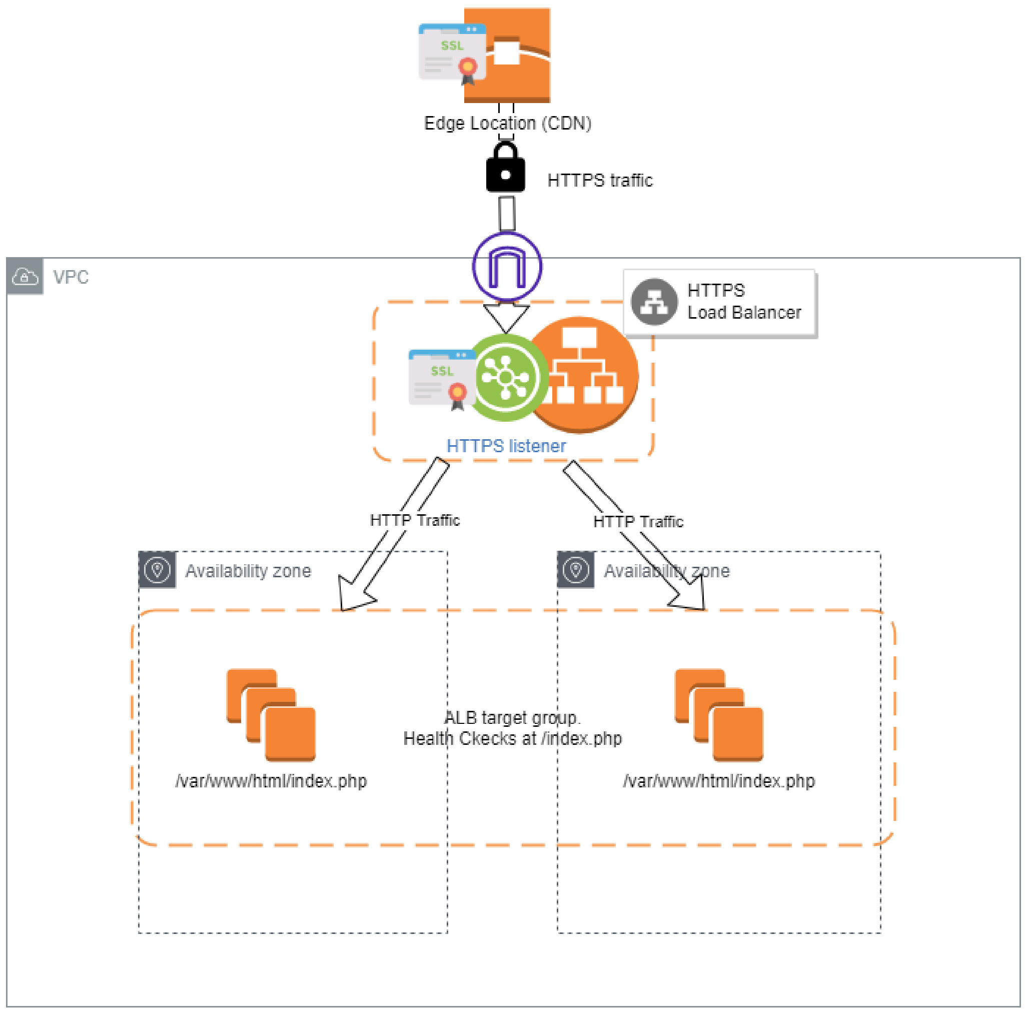


Figure 5.5: ALB with SSL offloading

5.5 Auto Scaling

AWS Auto Scaling is essential for optimizing costs and improve overall performance. With this service the user ensures that always has the required compute capacity at any time of the day without having to commit in advance buying lots of infrastructure trying to guess which will be the maximum workload. In essence, with autoscaling, applications will always have the right resources at the right time.

A launch template or a launch configuration (two pretty similar AWS services) are used to soecify the software and hardware configuration (type of instance, GB of storage, ssh key, OS image or AMI, etc) of the instances that will be launched by an autoscaling group.

Thanks to Auto scaling, multiple AWS services, such as EC2, can scale horizontally (or outside) meaning that more instances can be added or shut down automatically depending on certain parameters, like the % of CPU usage.

To achieve that purpose AWS introduces some theoretical concepts:

- Minimum size: minimum number of instances allowed
- Maximum size: maximum number of instances allowed
- Desired capacity: number of instances that will be launched at boot time and that will be maintained if possible.
- CloudWatch alarms: rules that are used to determine when to create or terminate instances based on some metrics and threshold values.
- Auto scaling policies: are associated with cloudwatch alarms and indicate how the auto scaling has to be performed. For example, a change in capacity, only 1 by 1 and with a cooldown of 60 seconds.

In my project I configured two auto scaling groups:

- The first one is for the bastion hosts. Bastion hosts are EC2 instances that can only be accessed through SSH by a specified IP address (thanks to the bastion host security group) and thanks to an SSH agent are able to connect the user with the instances located in private subnets. Bastions are normally used for troubleshooting purposes.

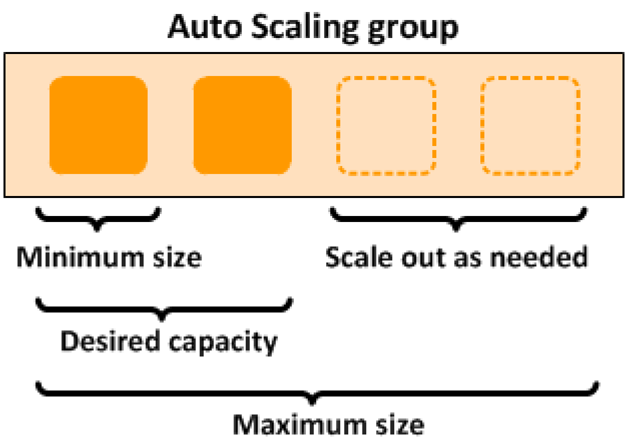


Figure 5.6: AWS Auto Scaling. Source: docs.aws.amazon.com

The aim of this auto scaling group is to ensure that one and just one bastion host instance is deployed in the public subnet of each availability zone. Having in mind there are only 2 AZs, the maximum size will be equal to 2, which also will be equal to the minimum size and desired capacity. No more instances can be instantiated and if an instance for some reason terminates, another instance will be spinned up automatically. No auto scaling policies are needed.

- The second one is much more complex. The aim of this auto scaling group is to ensure that the general CPU usage of the private instances is always maintained between two % defined by the user via the GUI, let's say 80% and 20%, while the maximum or minimum number of instances, both also defined by the user (for example, 2 and 4, respectively), is not surpassed. In this case, there is no desired capacity specified in the code, when this happens, the desired capacity is equal to the minimum size.

  Let's say there are 3 instances running and the average CPU usage has been, for more than two periods of 120 seconds, greater or equal than 80%. Given that case, the CloudWatch alarm is triggered and the scale up autoscaling policy is called. Then, it proceeds to perform an scaling adjustment of +1 to the number of instances and applies it plus a cooldown of 60 seconds.

  Now that there are 4 instances, the CloudWatch alarm will not be triggered anymore because the maximum number of instances has been reached. This behaviour is equal when scaling down: if the CPU usage is systematically below 20% and the minimum number of instances has not been reached, one instance is automatically terminated.

5.6 SSL-TLS Certificate

ACM or AWS Certificate Manager is an AWS service that provisions, manages and deploys SSL/TLS certificates. This service removes the tim-consuming task of having to purchase and yearly renew SSL/TLS certificates. ACM can be used with Elastic Load Balancing, Amazon Cloudfront, Amazon API Gateway and few other resources.

In my project I created two certificates. One for the AWS CDN, which I will explain in short, and one for the application load balancer.

The reason why I deploy two certificates instead of just the CDN one is that instead of using the CDN to do the SSL offloading, the ALB is the one that does it and thanks to that HTTPS is used between Cloudfront and the ALB and the traffic in transit is encrypted a security feature sometimes required by companies.

When created, both certificates require some sort of validation. Email validation is easier to implement but cannot be automatized because it requires human intervention. That is why I chose to use DNS record validation, which is also done automatically thanks to terraform.

DNS validation consist of adding a CNAME record with a unique key-value pair that demonstrates that the user owns the domain. This is done for both certificates.

Additionally, the Cloudfront certificate has to be created in the us-east-1 region in order to be seen by the CDN. This is normal because Cloudfront is a global service, not a regional service. This is why in my Terraform configuration, as can be seen in figure 6.4, I need to add another provider block with the us-east-1 region and alias. I need to ensure that no matter which region the user choses, there has to be a provider in that region.

5.7 Route 53 (DNS)

Route 53 is a high available and scalable DNS AWS service. It translates domain names into IP addresses in a very cost effective way. In my platform I only offer the possibility of using an AWS registered domain name.

There is not the possibility of using a domain name that has not been registered or is not managed by AWS because of the SSL DNS validation mode for verifying that the ACM certificates are usable. Because of this, all domains that use an external DNS (the ones that have not been issued by AWS) cannot be verified through this process automatically and an error is returned.

When a domain name is created or mainained by AWS, a hosted zone is automatically created with an NS record and a SOA record. Then, other records can be added. Hosted zones serve the purpose of managing the records of the domain name. The following records are the most important that can be added and the ones I use in the project:

- Alias: Used by naked (apex) domains or other records in the hosted zone to reference the DNS names of ELB, Cloudfront distributions or S3 buckets configured as websites. They work like CNAME records work for FQDN domain names.
- A (Adress): Given a domain name an A record translates it to a IP.
- SOA (Start of Authority): Every DNS address begins with this record. That is why is automatically created when a domain name is registered with AWS. It stores information about the server that supplied the data for the zone, the TTL (time to live: time the record is cached in own PC or resolving server), version, admin of the zone, etc.
- NS (Name Servers): Used by TLD servers to direct traffic to the content DNS server that contains the DNS records.
- CNAME (Canonical Name) : Record that matches one FQDN (Fully Qualified Domain Name x.y.com) to another FQDN z.y.com

In my project, apart from the SOA and NS records that are already created and the DNS validation of the ACM certificates, I create an A record for the apex domain name which I point it to the ALB DNS name, and a CNAME for the www FQDN of the apex domain name. Other subdomains and DNS routing strategies are up to the user once the infrastructure is deployed.

5.8 Cloudfront

Cloudfront is the CDN service of AWS. A CDN is used for serving static content in a secure way with less latency and improving overall performance. When a CDN is created, it is called a distribution, and its neuralgic point is the origin.

The origin of a Cloudfront distribution is the location where the content is stored and from which CloudFront gets the content to serve. the origin can be a server, an S3 bucket, a load balancer, etc

As explained in section 5.1, content is delivered and cached through edge locations. If the request's content is not already cached, the request itself is forwarded to the origin

Using Cloudfront is better even for dynamic content (the one that is not cached) because it helps performing faster TLS negotiations and routes requests to the origin using the AWS global backbone network instead of using the public internet.

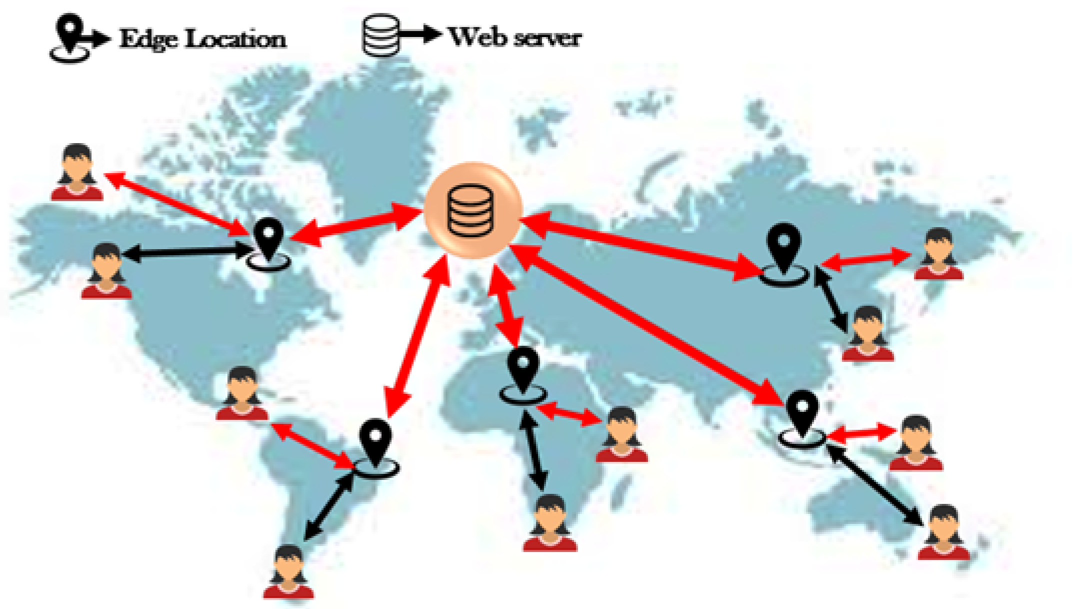


Figure 5.7: AWS CloudFront. Source: javatpoint.com

In my project I create an amazon Cloudfront with a custom origin: the ALB. The origin is a HTTPS ALB and listens to the 443 port in https-only mode.

This distribution also has a WAF attached, that will be explained in the following section. In order for the users to correctly access the distribution without using the CloudFront DNS name it has an alias, which corresponds to the domain name.

The default behaviour of the distribution is to catch only the GET and HEAD requests. In addition, all the other HTTP methods are allowed (DELETE, POST, PUT, etc), all cookies are forwarded, the TTL of the cached content in the edge locations is set to 1 hour and the viewer policy is set to redirect-to-https, which means that if the users search the domain name in HTTP, the request is automatically set to HTTPS.

However, the most special feature of my implementation is the X-Origin-Token custom header. All the requests that go through the CDN have a custom header called X-Origin-Token with a random string as a value and in front of the ALB there is another firewall that only accepts requests with this header.

Thanks to that, only requests that come from the CDN will be allowed. If somehow someone manages to get the DNS name of the ALB, the only response that will get is a HTTP 403 error (Forbidden).

I got this idea from the Core Solutions blog example [18].

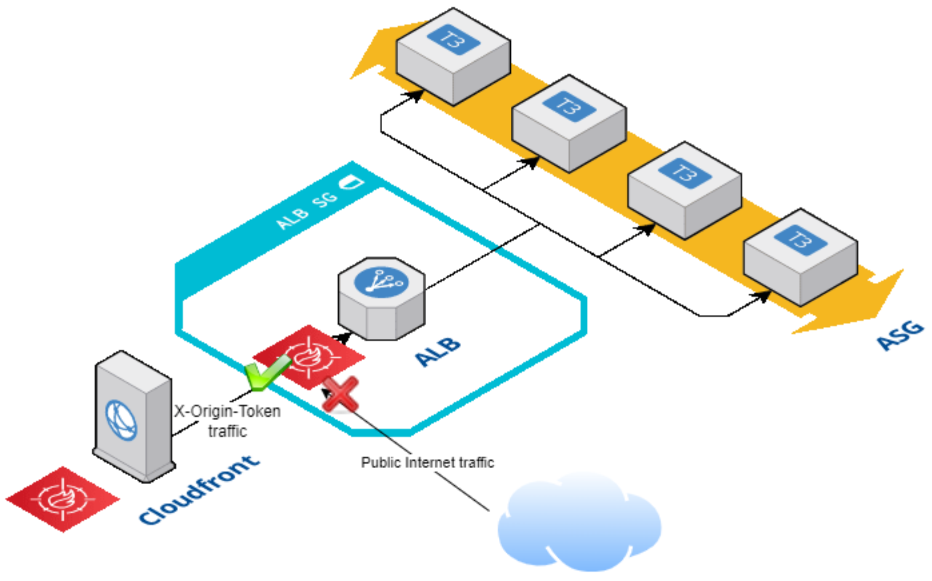


Figure 5.8: X-Origin-Token.

5.9 AWS WAF

Despite having AWS security groups, sometimes there is the need for a more explicit security. AWS Web Application Firewall helps protecting applications against common web attacks such as SQL injection, cross-site scripting(XSS) and a large etcetera.

WAFs [6] are pretty similar to Network ACLs in the sense that they also work with rules. This rules can be managed by AWS or customized by the user.

In my deployment I use two Web Application Firewalls:

- CloudFront WAF: The first one is in charge of protecting each single edge location where the static content is cached from OWASP top 10 security risks and bots. It also includes a IP blacklist and a request per time limit. All this features are thanks to AWS managed rules, a list of rules created and maintained by AWS itself.

  This WAF has to be created in the eu-east-1 region in order to be seen by the CloudFront distribution.

- ALB WAF: Thanks to that firewall, the ALB can only accept requests that come from the CloudFront CDN and include the X-Origin-token, as explained in the previous section.

  I created this one using a terraform module developed by core solutions [25].

5.10 Databases

AWS offers a wide variety of database services. From relational to noSQL, in-memory, document or grpah databases. In my deployment I focus on relational databases.

AWS Relational Database Service offers 7 different engines. I offer RDS MySQL and Aurora databases. AWS RDS offers scalable, fully managed relational databases

- RDS MySQL: It is a fully managed MySQL database on the cloud that offers high availability thanks to its multi-AZ deployments. In my deployment I create a multi-AZ deployment and user can decide if backups are taken, if the content gets encrypted at rest, how much storage can the database have and the CPU and RAM of its instances. [2]

- Aurora: Aurora is MySQL and PostgreSQL compatible. It offers 5x and 3x better performance than normal MySQL and PostgreSQL setups, respectively. To stay consistent with the RDS option I also use MySQL.

  It is the most powerful database option. It offers 99.99% uptime SLA, multi-AZ availability, continuous backups, built-in security, etc. And all that while maintaining really low costs compared to the commercial databases.

  Aurora has its compute layer completely detached from its storage layer. The storage layer spans across 3 AZs and each AZ has 2 complete copies of all the data. So, in total 6 copies of data are synchronously replicated.

  Aurora really gets complicated with clusters of clusters, cross-region clusters, read replicas of read replicas, etc, but for my deployment I just use one cluster of database instances with one writer instance and from 1 to 15 reader instances.

  Its RAM and CPU power can be specified by the user. Auto scaling for reader instances is enabled, which fits perfect for read intensive applications like WordPress.

  For deploying the Aurora infrastructure I make use of another Terraform module [7]

Database credentials need to be created before the deployment. That avoids the leakage of sensitive information in public version control repositories. This topic is explained in the beginning of chapter 6.

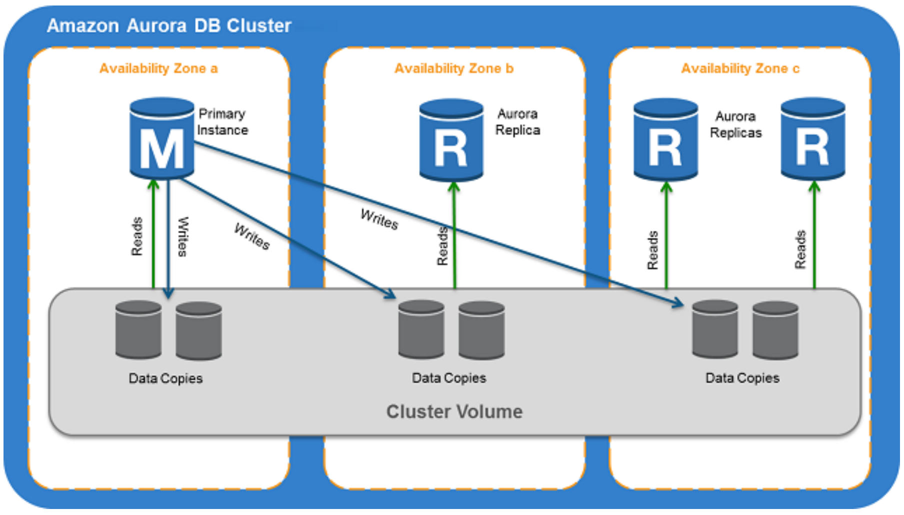


Figure 5.9: Aurora Cluster. Source: docs.aws.amazon.com

5.10.1 Elasticache

Elasticache is a web service that makes it easier to launch, manage and scale a distributed in-memory database cache in the cloud. It can provide up to microsecond latency! really important for live streaming and online games.

In my deployment I offer both options that AWS supports: Redis and Memcached engines [21].

With the Memcached deployment the user can decide the RAM and CPU power of the Elasticache instances and the number of read replicas, which is up to 5. It uses the port 11211.

The Redis deployment can be in cluster mode or without cluster mode [20]. The latter just offers one cluster (called shard), with one write or primary node and up to 5 read replicas. With cluster mode enabled there can be up to 500 shards, each one with up to 5 read replica nodes and can have data partitioning.

Memcached is easier to set up and is recommended for basic installations, however Redis offers more configuration options. Both of them offer pretty awesome and similar performances.

The table on this website summarizes pretty well the differences between Memcached, Redis (cluster mode disabled) and Redis (cluster mode enabled) [19]

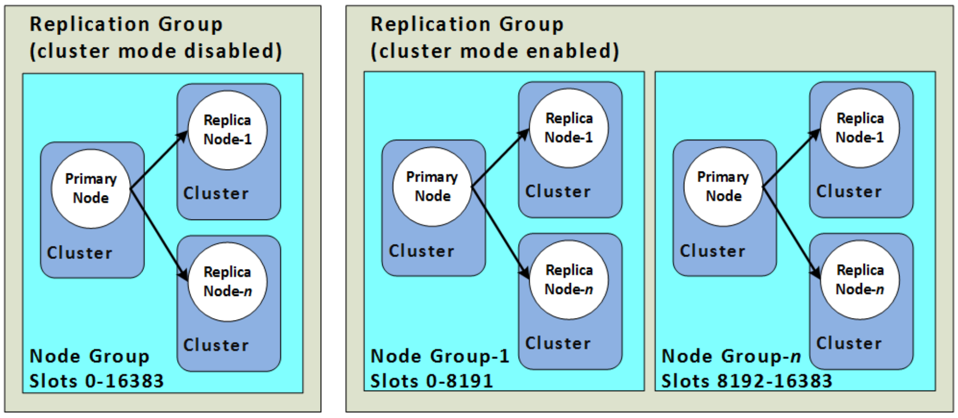


Figure 5.10: Redis Cluster mode enabled vs disabled. Source: docs.aws.amazon.com

5.11 EFS

There is a crucial need of mounting a shared storage in this configuration because there are many web applications and services, such as WordPress, that are stateful, meaning that if an instance shuts down (due to auto scaling reasons) it could lose valuable information that could never be recovered.

Imagine that an admin updates a post and the instance is terminated. This post will never be seen again. Instances need to preserve the state and act as a team that works together rather than multiple teams.

EFS service is exactly that, a scalable and fault tolerant storage point that can be mounted by thousands and thousands of servers without network overhead.

With this shared mount point, instances can automatically mount it to the desired folder [17] (configuring /etc/fstab) as if it was one more piece of local storage, with the difference that any modification will affect all the other instances that also mounted it.

The storage capacity automatically scales up and down and user is charged for every byte stored.

In my project I create one mount target in each AZ of my infrastructure and I locate them in the database private subnets. There is no problem with other public and private subnets in the same AZ, as the infrastructure inside them will be able to access it. As the AWS official documentation explains ”You can create only one mount target per Availability Zone. If an Availability Zone has multiple subnets, as shown in one of the zones in the illustration, you create a mount target in only one of the subnets. As long as you have one mount target in an

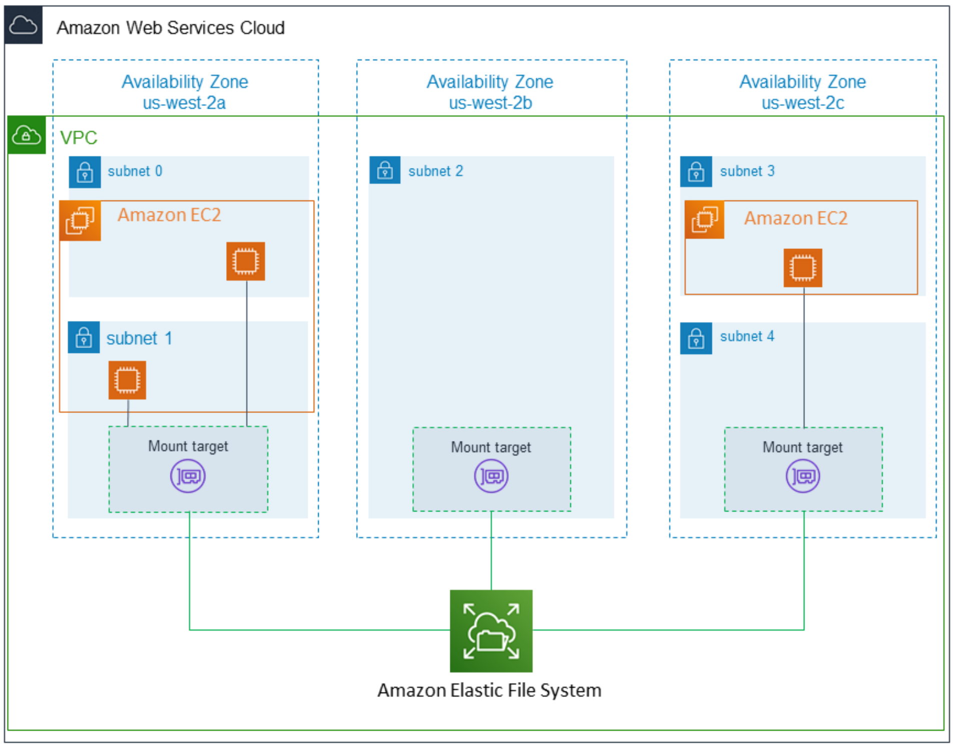

Amazon Web Services Cloud
Availability Zone us-west-2a
Availability Zone us-west-2b
Availability Zone us-west-2c
VPC
subnet 0
subnet 1
subnet 2
subnet 3
subnet 4
Amazon EC2
Amazon EC2
Mount target
Mount target
Mount target
Amazon Elastic File System

Figure 5.11: EFS mount targets

Availability Zone, the EC2 instances launched in any of its subnets can share the same mount target." [17]

Contents stored that are stored in the EFS unit but have not been accessed for more than 30 days will be moved to the infrequent access storage tier, which is a cheaper tier with a longer retrieval time. Once they are accessed again, contents will automatically be moved to the primary storage class.

5.12 AWS Tags

Tags are metadata used to easily identify, organize, manage and even filter AWS services. Each tag consists of a key and a value. In my project I defined 3 tags per resource.

- Name: The name of the resource. All the names contain project and environment information along with the name of the resource.
- Project: Name of the project.
- Environment: Environment in which the infrastructure has been deployed. It can be dev or prd.

Thanks to terraform, applying AWS tags has been easier than ever. I define a locals block with a tags block that contains the Project and Environment keys equal to the corresponding variables, as can be seen in Figure 4.6.

Chapter 6

Guest User Interface

In order to make the infrastructure configuration simpler, faster, more robust and intuitive I created a GUI. It is designed using python and pySimpleGUI, a Python package that easily enables the creation of simple GUI designs for Windows, Linux and Mac, which fits perfectly for this purpose.

It consists of a ”form” that includes all the variables needed to set up the infrastructure. Once all the variables are included, the create button can be pressed and then the GUI automatically performs some steps that will be explained in detail in the following sections. In short, it creates some terraform files, launches several scripts, creates directories, repositories and automatically launches the AWS infrastructure.

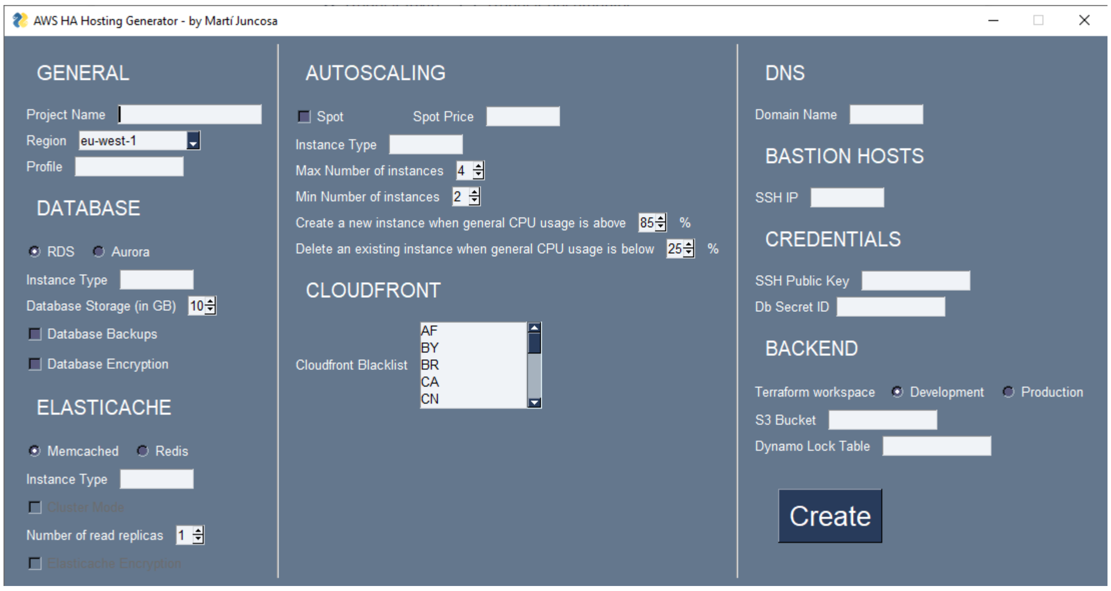
Apart from creating the infrastructure, this GUI is intended to set up the bases of a CI/CD environment where a team can work simultaneously in the same project in a fast and reliable pace.

The form has 6 different input types: text fields, radio buttons, check boxes, spin boxes, combo boxes and data list boxes. For further understanding, below I explain every single input:

- General

  Project name: The project name will be applied to the name of each AWS resource that is created. It is also the name of the local directory and the name of the GitHub remote repository.

  Region: The AWS region in which the infrastructure will be created. There are 22 regions spread across the world available in normal AWS accounts. It is important to have


AWS HA Hosting Generator - by Martí Juncosa

GENERAL

Project Name
Region: eu-west-1
Profile

DATABASE

[✓ RDS] Aurora
Instance Type
Database Storage (in GB): 10
☐ Database Backups
☐ Database Encryption

ELASTICACHE

[✓ Memcached] Redis
Instance Type
☐ Cluster Mode
Number of read replicas: 1
☐ Elasticache Encryption

AUTOSCALING

☐ Spot
Spot Price
Instance Type
Max Number of instances: 4
Min Number of instances: 2
Create a new instance when general CPU usage is above 85 %
Delete an existing instance when general CPU usage is below 25 %

CLOUDFRONT

Cloudfront Blacklist: AF BY BR CA CN

DNS

Domain Name

BASTION HOSTS

SSH IP

CREDENTIALS

SSH Public Key
Db Secret ID

BACKEND

Terraform workspace: [✓ Development] Production
S3 Bucket
Dynamo Lock Table

Create

Figure 6.1: Preliminary design of the UI

this parameter in mind because it can impact directly on pricing and user latency.

Profile: The profile is needed in the main.tf file and it is a way of defining AWS access credentials without having them in code. Which is mandatory if the project has some sort of version control. With the aim of standardizing the platform I have defined a unique path for this item, that must be located in /.ssh/credentials. [15]

- Database

  RDS/Aurora: There is the possibility of choosing between AWS RDS MySQL or Aurora. Differences between them are explained in section 5.10.

  Instance type: RDS instance types must have a prefix of db. For example db.t3.micro. IF the user choses Aurora the minimum instance type supported is a db.t3.small. It indicates the RAM and CPU power. User needs to make sure the AWS nomenclature is followed.

  Database Storage: If using RDS, this field indicates the database storage in Gigabytes. In case of fulfilling all the storage, RDS automatically scales up until reaching the maximum database storage, which in my platform I have defined it to be 10x times the base database storage.

  Database Backups: Enable database backups. Aurora manages them itself but RDS, if enabled, maintains a copy during 14 days and each backup is taken between 3:15 and 4:15 am.

  Database Encryption: Enables or disables database content encryption at rest.

- Elasticache

Memcached/Redis: AWS offers just two types of in-memory database caches. Both options are explained in detail in section 5.10.1.

Instance type: Elasticache instance types must have a prefix of cache. For example cache.t3.micro. It indicates the RAM and CPU power. User needs to make sure the AWS nomenclature is followed.

Cluster mode: Cluster mode only works with Redis. It enables the user to have two primary databases (instances that read and write), each one with its read replicas.

Number of read replicas: The number of read-only nodes each cluster will have. It works with just one cluster as well. If the application to be deployed is read intensive this option should be considered.

Elasticache encryption: Only supported by Redis. Encryption is at rest, not in transit.

- Autoscaling

  Spot: Spot instances benefit from unused EC2 capacity in the AWS cloud and can reduce up to 90% of your on-demand price. However they can be removed at any time, so it is not recommended for constant stateful workloads that cannot be interrupted.

  Spot price: Spot instances are purchased through bids. As the demand is fairly constant the prices remain the same and can be easily searched through google. Remember that if someone outbids you, your AWS instances will be removed from your account.

  Instance type: Instance types do not need a orefix. For example t3.micro. It indicates the RAM and CPU power. User needs to make sure the AWS nomenclature is followed.

  Max number of instances: Upper threshold of the permitted autoscaling instances. Even if all instances are in 100% CPU usage no more instances will be instantiated.

  Min number of instances: Lower threshold of the permitted autoscaling instances. Even if all instances are in 0% CPU usage no more instances will be deleted.

  Create a new instance when general CPU usage is above: This field allows the user to define the trigger point of the policy that instantiates another instance based on the general CPU usage.

  Delete an existing instance when general CPU usage is below: This field allows the user to define the trigger point of the policy that deletes another instance based on the general CPU usage.

- Cloudfront

  Cloudfront blacklist: Offers the possibility of restricting access to a specific country. The country names are introduced following the ISO 3166 country codes because those are the ones accepted by the AWS provider of terraform. The countries in the list are just the most georestricted ones, there are not all the countries.

- DNS

  Domain name: The domain name must be registered within AWS. If it has been registered with another registrar make sure that AWS supports the TLD of your domain name and transfer it to AWS before using this platform.

- Bastion Hosts

  SSH IP: The only public IP that will be allowed to connect via SSH to the bastion hosts, and thus, via an SSH agent, be able to connect to the private servers. In the GUI, it has to be introduced without a mask, only the IP: 1.2.3.4.

- Credentials

  SSH public key: This public key will be stored inside the bastion hosts and thanks to your private ssh key from the key pair, the SSH connection will be possible. It must be located in the default folder: /.ssh/.

  Db secret ID: This secret pair must be created and stored with AWS Secrets Manager service. In addition, the pair must have two values, each one with its key (or identifier), and value. The keys must be named username and password respectively, both values are up to you and will be used as the username and password of your database, either if you chose to use RDS MySQL or Aurora.

- Backend

  Terraform workspace: Terraform workspaces makes possible for the backend to store multiple states with a single terraform project. I propose two workspaces: dev and prd for the development and production working enviroment respectively.

  S3 bucket: The AWS S3 service will be in charge of storing the terraform state file of each project deployed with this platform. This bucket must be created beforehand.

  DynamoDB lock table: As explained in the terraform chapter, this noSQL database is in charge of locking out the terraform state file of the S3 bucket when it is being modified (this normally happens when a deployment is being done) in case other people want to modify it simultaneously. This item must also be created beforehand.

6.1 Dependencies

Although the complexity of the platform is not excessive there are some dependencies to satisfy before using the GUI. Normally, in an enterprise environment are already satisfied but for the average user may require some action.

The user must have:

- GitHub account
- AWS account

The following software must be installed

- Terraform 1.1 or higher
- Github CLI
- Git
- Python 3.9 or higher

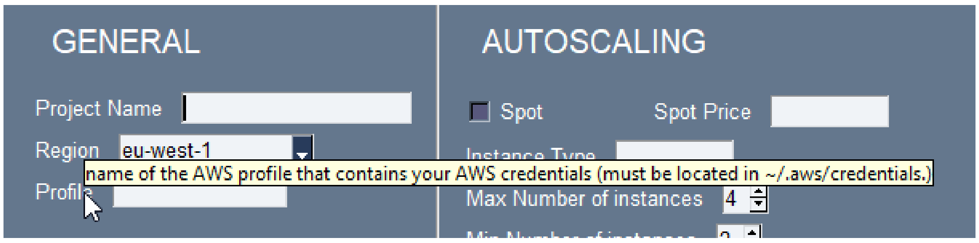
Actions before usage

- GitHub SSH key pair and GitHub CLI authentication
- AWS profile must be created in /.aws/credentials file
- AWS db credentials must be created using AWS secrets manager
- ssh key pair must be created inside /.ssh folder
- S3 bucket has to be created: it will store the terraform file state.
- BynamoDB table has to be created: for locking the state file in the remote backend when a user performs a terraform apply.

6.2 Features and security

Some input fields may be disabled depending on the resources selected. For example, as seen in figure 6.1, memcached engine is selected in the elasticache section. Because of that, as it does not support cluster mode and elasticache encryption, both check boxes are disabled. It happens the same with the database section: when Aurora is selected, database storage and database backups are automatically disabled because AWS Aurora manages them itself.

Every input that requires some explanation or hint has a tooltip explaining why is that field important and what the user should introduce.



eu-west-1

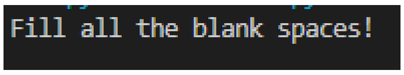
name of the AWS profile that contains your AWS credentials (must be located in ~/.aws/credentials.)

4

Figure 6.2: Tooltip

It also has some security features to prevent accidental deployments. All the text fields except the spot instance field, which is optional, must be fulfilled before pressing the create button. Otherwise this action is not allowed and a disclaimer appears.

Spot price is only considered if spot mode is enabled. Cloudfront blacklist can be null, there can be no georestricted countries selected.



```
Fill all the blank spaces!
```

Figure 6.3: Disclaimer indicating that no or not all the text fields have been fulfilled

In addition, all the spin boxes have lower and upper thresholds. Number of read replicas has a max value of 5, because it is the max number of read replicas permitted for a in-memory cache cluster, and a lower value of 0.

The autoscaling instances spin boxes have a maximum of 10 and a minimum of 2. In addition the min number of instances can not be greater than the maximum number of instances and vice versa. Finally, the autoscaling scaling up and scaling down policies depending on the CPU usage have thresholds of 50% to 95% and 5% to 49% respectively.

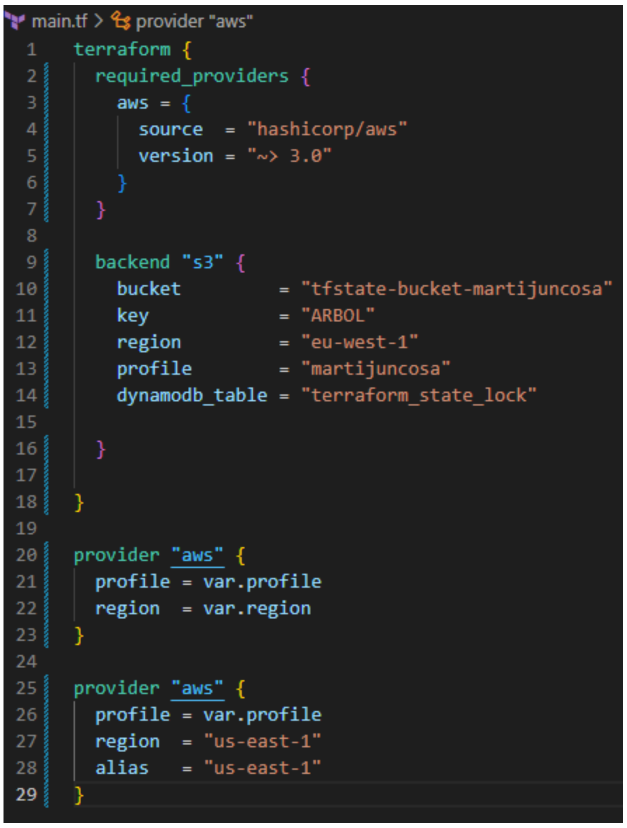
6.3 Step 1: create the main.tf file

After introducing all the variables and pressing the create button the first step that the platform does is to create the main.tf file.

This file is a terraform file that contains the terraform block with the required provider definition and the backend definition along with the provider block definitions.

While it seems pretty easy to write, it is one of the most (if not the most) important files of the project because it specifies which provider should terraform use and where terraform does have to store the state file.

The reason why I create it dynamically with the GUI during running time and not before as


main.tf > provider "aws"

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }

  backend "s3" {
    bucket         = "tfstate-bucket-martijuncosa"
    key            = "ARBOL"
    region         = "eu-west-1"
    profile        = "martijuncosa"
    dynamodb_table = "terraform_state_lock"

  }

}

provider "aws" {
  profile = var.profile
  region  = var.region
}

provider "aws" {
  profile = var.profile
  region  = "us-east-1"
  alias   = "us-east-1"
}
```

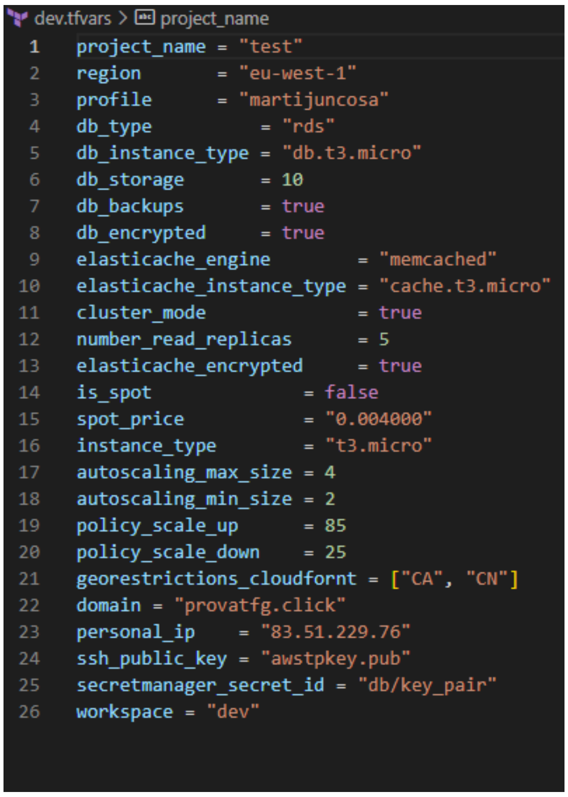
Figure 6.4: main.tf file

every other terraform file is because of the backend.

The backend block has to be hard coded, as can be seen in the figure 6.4, meaning that its different fields cannot be equal to a variable. It has been years since the terraform community started demanding a more feasible way of automating terraform with a remote backend, but as far as I know this is the only way for now. [8]

6.4 Step 2: create tfvars file

Creating the tfvars file is the second step. Its name will be dev.tfvars or prd.tfvars depending on the workspace selected. From now on it will be referred as {workspace}.tfvars. This file contains all the information introduced by the user and configures the rest of the project.


dev.tfvars > project_name

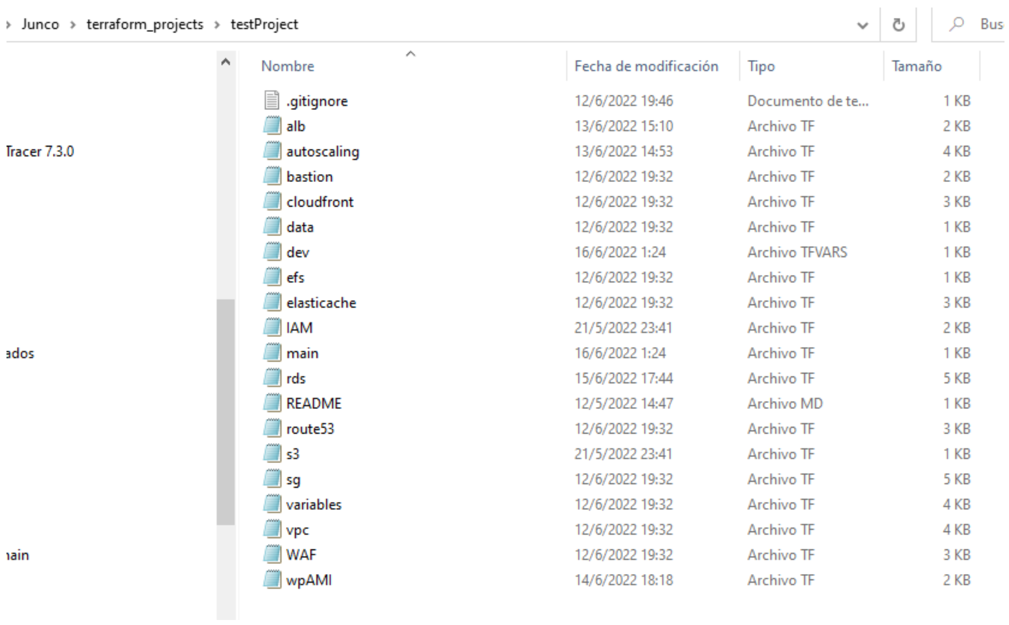
```
project_name = "test"
region       = "eu-west-1"
profile      = "martijuncosa"
db_type            = "rds"
db_instance_type = "db.t3.micro"
db_storage         = 10
db_backups         = true
db_encrypted       = true
elasticache_engine          = "memcached"
elasticache_instance_type = "cache.t3.micro"
cluster_mode                = true
number_read_replicas        = 5
elasticache_encrypted       = true
is_spot                = false
spot_price             = "0.004000"
instance_type          = "t3.micro"
autoscaling_max_size = 4
autoscaling_min_size = 2
policy_scale_up        = 85
policy_scale_down      = 25
georestrictions_cloudfornt = ["CA", "CN"]
domain = "provatfg.click"
personal_ip     = "83.51.229.76"
ssh_public_key = "awstpkey.pub"
secretmanager_secret_id = "db/key_pair"
workspace = "dev"
```

Figure 6.5: dev.tfvars file

This file is created by concatenating and treating all the string inputs provided.

6.5 Step 3: create a local directory

From now on, all the steps are performed by a batch script called project_init.bat. The next step is to create a local directory under home/terraform_projects/. This repository contains all the necessary terraform files for the future maintenance or destruction of the project

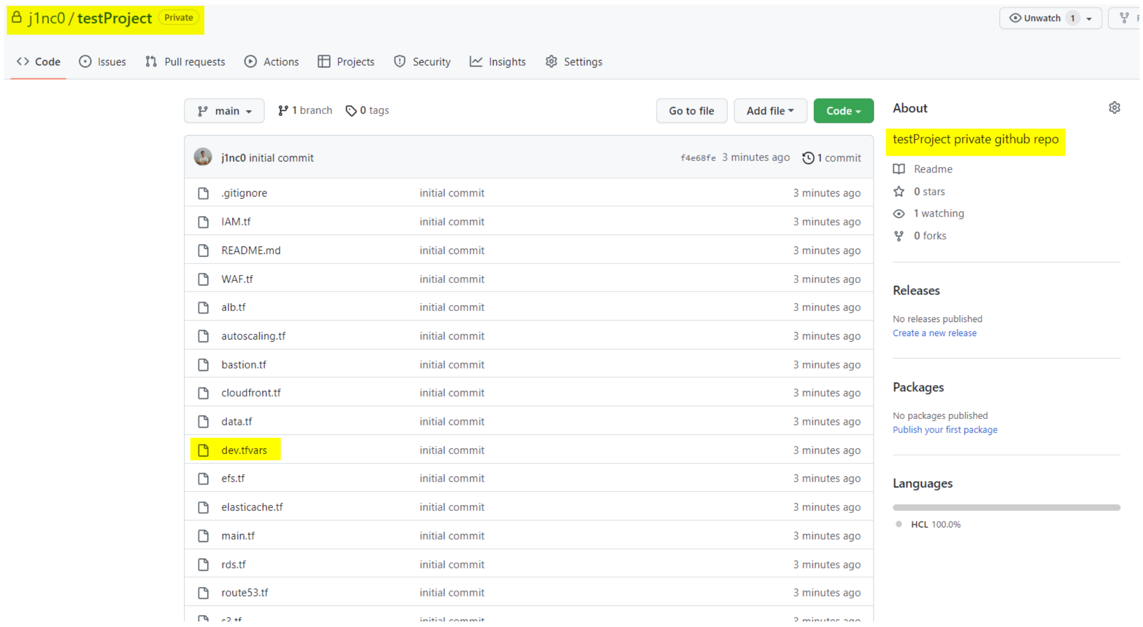


Nombre	Fecha de modificación	Tipo	Tamaño
.gitignore	12/6/2022 19:46	Documento de te...	1 KB
alb	13/6/2022 15:10	Archivo TF	2 KB
autoscaling	13/6/2022 14:53	Archivo TF	4 KB
bastion	12/6/2022 19:32	Archivo TF	2 KB
cloudfront	12/6/2022 19:32	Archivo TF	3 KB
data	12/6/2022 19:32	Archivo TF	1 KB
dev	16/6/2022 1:24	Archivo TFVARS	1 KB
efs	12/6/2022 19:32	Archivo TF	1 KB
elasticache	12/6/2022 19:32	Archivo TF	3 KB
IAM	21/5/2022 23:41	Archivo TF	2 KB
main	16/6/2022 1:24	Archivo TF	1 KB
rds	15/6/2022 17:44	Archivo TF	5 KB
README	12/5/2022 14:47	Archivo MD	1 KB
route53	12/6/2022 19:32	Archivo TF	3 KB
s3	21/5/2022 23:41	Archivo TF	1 KB
sg	12/6/2022 19:32	Archivo TF	5 KB
variables	12/6/2022 19:32	Archivo TF	4 KB
vpc	12/6/2022 19:32	Archivo TF	4 KB
WAF	12/6/2022 19:32	Archivo TF	3 KB
wpAMI	14/6/2022 18:18	Archivo TF	2 KB

Figure 6.6: Local directory

6.6 Step 4: create a GitHub repository

At the same time, a GitHub private remote repository is created with the same files as in the local directory. Notice that the repository name is the same as the local directory and that only the dev.tfvars file is present (only the dev workspace has been deployed).



j1nc0 initial commit f4e68fe 3 minutes ago 1 commit

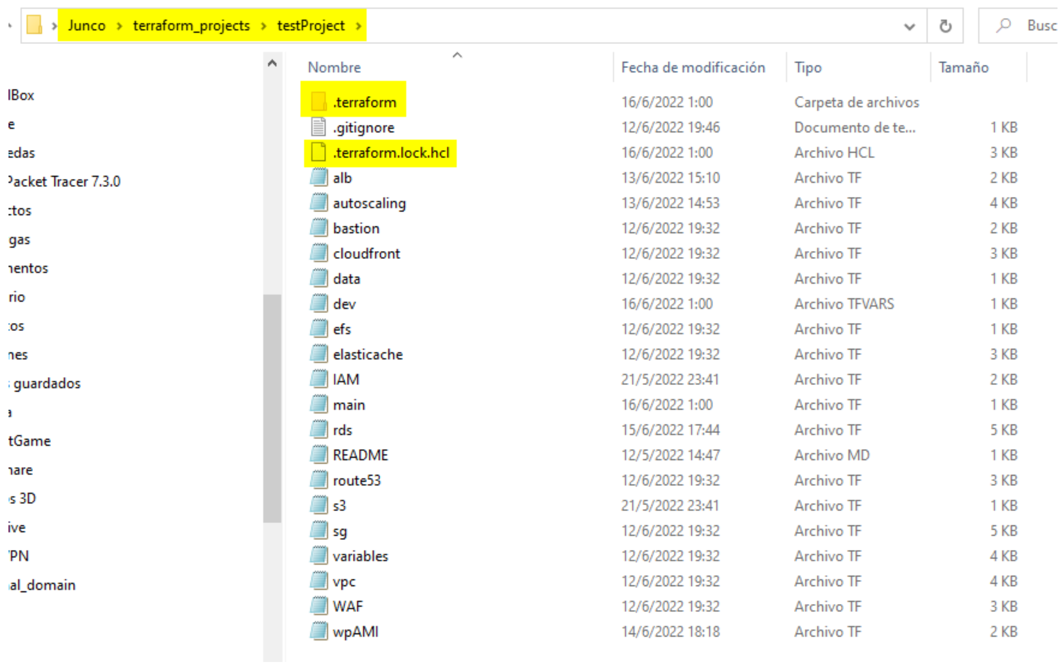
.gitignore	initial commit	3 minutes ago
IAM.tf	initial commit	3 minutes ago
README.md	initial commit	3 minutes ago
WAF.tf	initial commit	3 minutes ago
alb.tf	initial commit	3 minutes ago
autoscaling.tf	initial commit	3 minutes ago
bastion.tf	initial commit	3 minutes ago
cloudfront.tf	initial commit	3 minutes ago
data.tf	initial commit	3 minutes ago
dev.tfvars	initial commit	3 minutes ago
efs.tf	initial commit	3 minutes ago
elasticache.tf	initial commit	3 minutes ago
main.tf	initial commit	3 minutes ago
rds.tf	initial commit	3 minutes ago
route53.tf	initial commit	3 minutes ago

Figure 6.7: Remote repository

6.7 Step 5: terraform actions

The last action is to actually execute the infrastructure deployment. To achieve that, the terraform workflow pipeline is executed through the following commands:

- terraform init -upgrade: this command initializes the terraform working directory and upgrades the modules of the project thanks to the upgrade option.

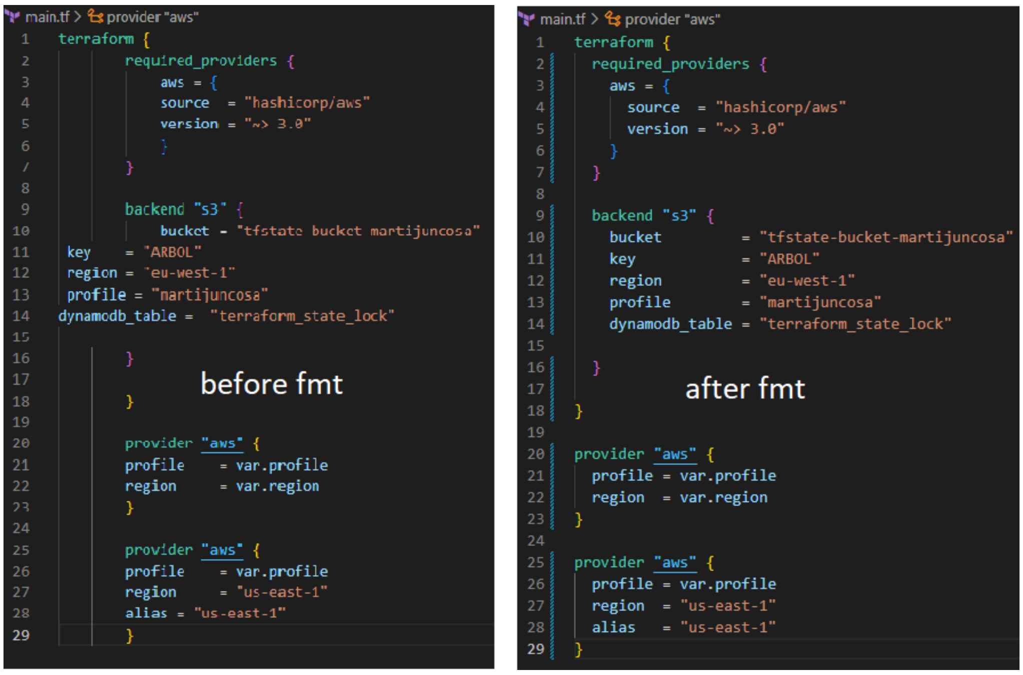


Nombre	Fecha de modificación	Tipo	Tamaño
.terraform	16/6/2022 1:00	Carpeta de archivos	
.gitignore	12/6/2022 19:46	Documento de te...	1 KB
.terraform.lock.hcl	16/6/2022 1:00	Archivo HCL	3 KB
alb	13/6/2022 15:10	Archivo TF	2 KB
autoscaling	13/6/2022 14:53	Archivo TF	4 KB
bastion	12/6/2022 19:32	Archivo TF	2 KB
cloudfront	12/6/2022 19:32	Archivo TF	3 KB
data	12/6/2022 19:32	Archivo TF	1 KB
dev	16/6/2022 1:00	Archivo TFVARS	1 KB
efs	12/6/2022 19:32	Archivo TF	1 KB
elasticache	12/6/2022 19:32	Archivo TF	3 KB
IAM	21/5/2022 23:41	Archivo TF	2 KB
main	16/6/2022 1:00	Archivo TF	1 KB
rds	15/6/2022 17:44	Archivo TF	5 KB
README	12/5/2022 14:47	Archivo MD	1 KB
route53	12/6/2022 19:32	Archivo TF	3 KB
s3	21/5/2022 23:41	Archivo TF	1 KB
sg	12/6/2022 19:32	Archivo TF	5 KB
variables	12/6/2022 19:32	Archivo TF	4 KB
vpc	12/6/2022 19:32	Archivo TF	4 KB
WAF	12/6/2022 19:32	Archivo TF	3 KB
wpAMI	14/6/2022 18:18	Archivo TF	2 KB

Figure 6.8: Local directory initialized

Notice the .terraform folder and .terraform.lock.hcl file. Both of them are created during the terraform initialization and contain cached provider modules, which workspace is currently active and the last known backend configuration. This files are only present in the local directory, not in the GitHub repository.

- terraform fmt: rewrites the terraform configuration files to a canonical format and style. Having in mind that the main.tf and the workspace.tfvars file have been created concatenating strings in python, this command is needed to ensure code quality.

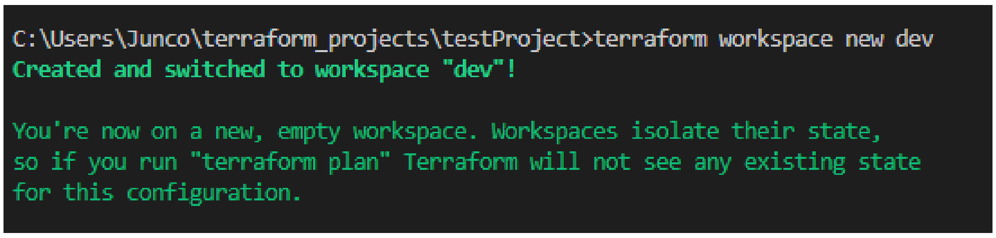


```
terraform {
        required_providers {
            aws = {
            source  = "hashicorp/aws"
            version = "~> 3.0"
            }
        }

        backend "s3" {
            bucket - "tfstate bucket martijuncosa"
 key     = "ARBOL"
 region = "eu-west-1"
 profile = "martijuncosa"
dynamodb_table =  "terraform_state_lock"

        }

        }

        provider "aws" {
        profile     = var.profile
        region      = var.region
        }

        provider "aws" {
        profile     = var.profile
        region      = "us-east-1"
        alias = "us-east-1"
        }
```

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }

  backend "s3" {
    bucket         = "tfstate-bucket-martijuncosa"
    key            = "ARBOL"
    region         = "eu-west-1"
    profile        = "martijuncosa"
    dynamodb_table = "terraform_state_lock"

  }
}

provider "aws" {
  profile = var.profile
  region  = var.region
}

provider "aws" {
  profile = var.profile
  region  = "us-east-1"
  alias   = "us-east-1"
}
```

Figure 6.9: terraform fmt command effect

- terraform workspace new {workspace}: creates a new workspace

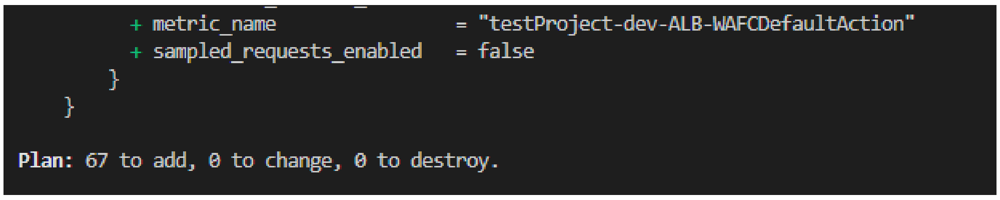


```
C:\Users\Junco\terraform_projects\testProject>terraform workspace new dev
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Figure 6.10: workspace command in cmd

- terraform workspace select {workspace}: selects the already created workspace

- terraform apply -var-file=workspace.tfvars -auto-approve: This command applies the current terraform configuration defined in the {workspace}.tfvars file and without prompting the user to approve it because of the auto-approve option. It is very dangerous to use if you are not a 100% sure that the outcome will be correct. In the figure below 67 AWS resources are being created.



```
          + metric_name               = "testProject-dev-ALB-WAFCDefaultAction"
          + sampled_requests_enabled  = false
        }
    }

Plan: 67 to add, 0 to change, 0 to destroy.
```

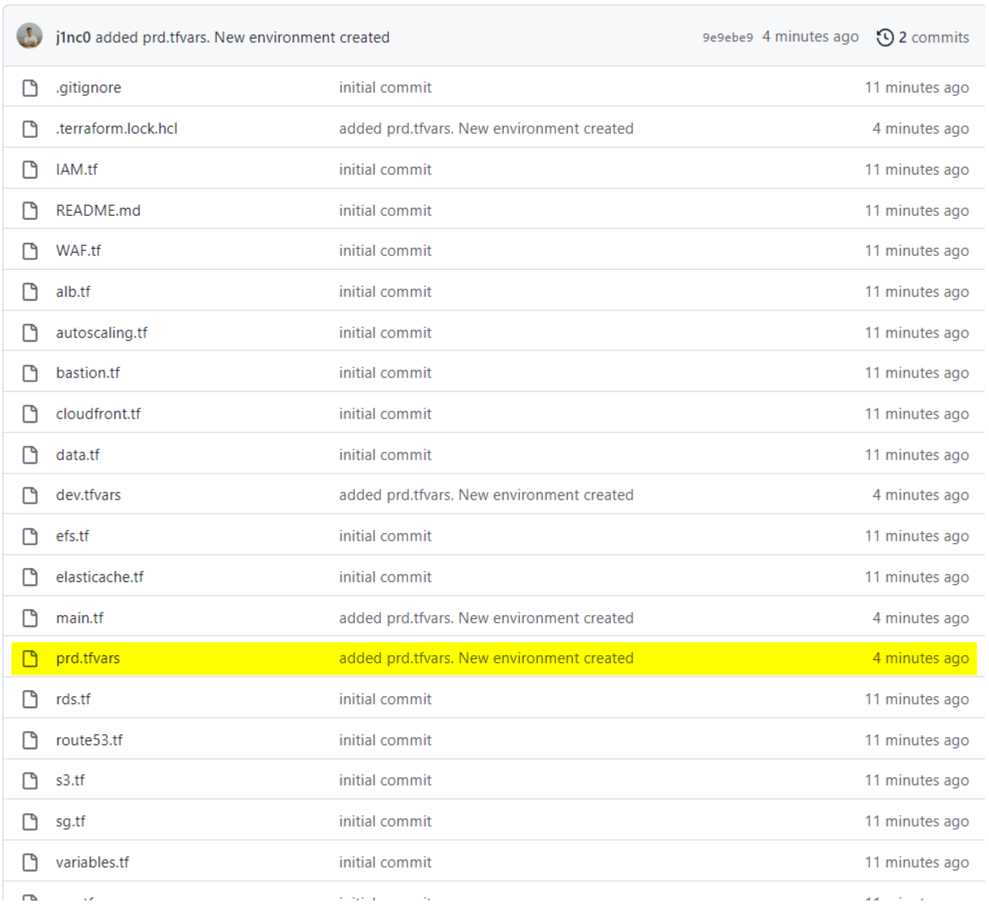
Figure 6.11: terraform apply command in cmd

6.8 Other project deployments and working with different workspaces

This process of 5 steps is repeated each time a new project is deployed. In order to deploy another project, the user must change the project name field in the GUI and whichever other parameter wants.

Then, a new local folder with the corresponding initialization will be created under home-/terraform_projects/, a remote Github repository will be created and the configured infrastructure will be deployed under the selected workspace.

However, the behaviour is a little different when deploying the other workspace of the same project. In that case, no local directory nor a remote GitHub repository should be created. Instead, just the new {workspace}.tfvars file is added to both of them; because they are already created by the first deployment.

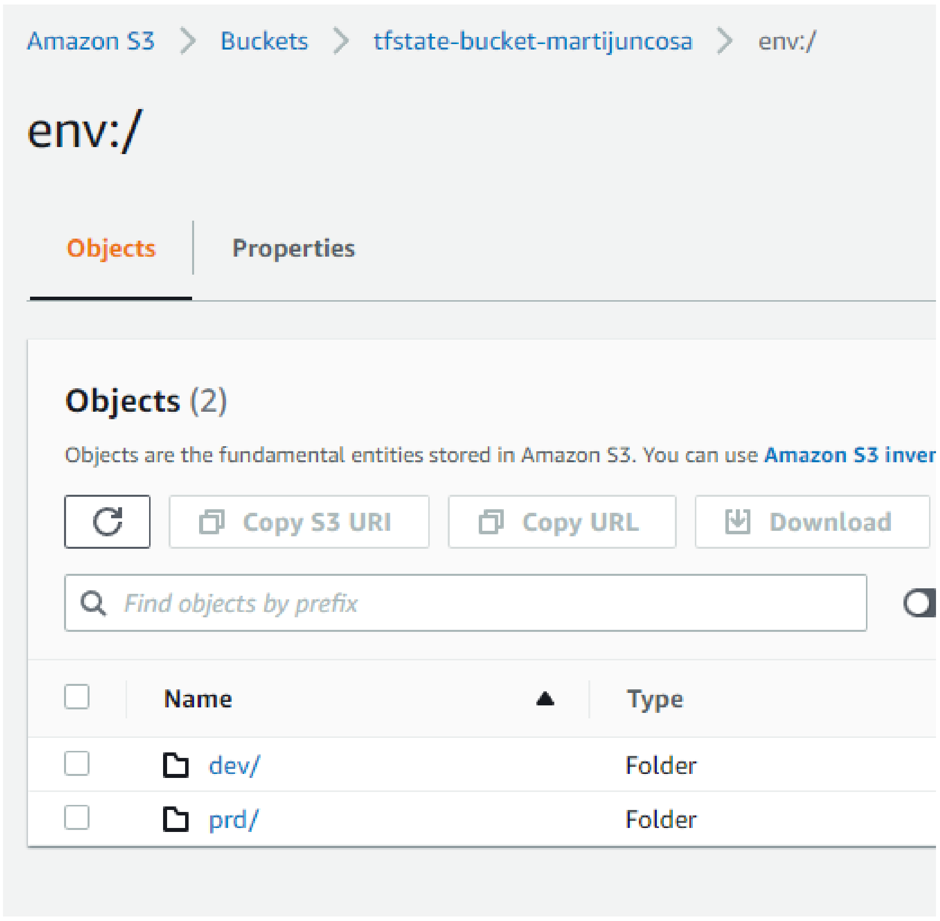


j1nc0 added prd.tfvars. New environment created   9e9ebe9 4 minutes ago   2 commits

.gitignore	initial commit	11 minutes ago
.terraform.lock.hcl	added prd.tfvars. New environment created	4 minutes ago
IAM.tf	initial commit	11 minutes ago
README.md	initial commit	11 minutes ago
WAF.tf	initial commit	11 minutes ago
alb.tf	initial commit	11 minutes ago
autoscaling.tf	initial commit	11 minutes ago
bastion.tf	initial commit	11 minutes ago
cloudfront.tf	initial commit	11 minutes ago
data.tf	initial commit	11 minutes ago
dev.tfvars	added prd.tfvars. New environment created	4 minutes ago
efs.tf	initial commit	11 minutes ago
elasticache.tf	initial commit	11 minutes ago
main.tf	added prd.tfvars. New environment created	4 minutes ago
prd.tfvars	added prd.tfvars. New environment created	4 minutes ago
rds.tf	initial commit	11 minutes ago
route53.tf	initial commit	11 minutes ago
s3.tf	initial commit	11 minutes ago
sg.tf	initial commit	11 minutes ago
variables.tf	initial commit	11 minutes ago

Figure 6.12: remote AWS backend with both workspaces

So, the file is added to the local folder, a new commit is pushed to the remote repository, the new workspace is created, the infrastructure is deployed and the terraform state file is stored in the corresponding workspace.



Objects (2)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inver

Name	Type
dev/	Folder
prd/	Folder

Figure 6.13: remote AWS backend with both workspaces

Chapter 7

Conclusions

After developing the project I can conclude that the final results have been utterly satisfactory and all the objectives and sub-objectives defined in chapter 1.3 have been achieved.

7.1 Objectives

The objectives were to create a software based tool that made the deployment of complex web hosting environments easier and quicker. I wanted the infrastructure to have certain resources and technicalities, but it has ended up being much bigger and complex than I expected, as can be seen in Figure 5.2.

It also includes a remote backend 6.7, two workspaces (development and production) 6.7, GitHub integration 6.6 and a simple and intuitive GUI developed with python 6. All those features combined provide a platform able to improve fast-paced team environments.

All the code developed can be checked in the public GitHub repository I created.

During this months, I signed up in Cloud Guru, a cloud learning platform, and I got 2 certificates that taught me the basics of the AWS cloud and Terraform. On March 17th I got the AWS Certified Cloud Practitioner and on April 4th I got the Hashicorp Certified: Terraform Associate accreditation.

This certifications along with my huge interest in the cloud opened the doors of the cloud industry for me and in April 27 I got a job as a cloud consultant in a cloud company based in Barcelona that works with AWS and Microsoft Azure public clouds.

Figure 7.1: AWS Certified Cloud Practitioner and Terraform Associate certifications

Furthermore, I will soon get examined of the AWS Certified Solutions Architect Associate, which is not a foundational certification anymore and will allow me to embrace in even more challenging projects.

Talking about time saving aspects, deploying this kind of infrastructure with the platform I have create can take, until it is completely set up and operative, 25 to 30 minutes. 5 to 10 minutes dedicated to correctly introduce variables and 15 to 20 dedicated to the automated deployment. It is not prone to errors and after the deployment, the infrastructure can be easily destroyed and managed.

If the same infrastructure had to be deployed in the AWS cloud without Terraform and my platform, it would take from 2 to 4 hours depending on the level of expertise of the engineer, it would be prone to human errors and it would not have a copy of everything that is deployed, and thus, if it had to be destroyed, it would have to be done by hand, which would add to the management process a considerable overhead.

In one of my test I lost the state file and I had to delete everything manually. Considering all the resource dependencies and that it was the first time I spent more than one hour deleting resources.

Finally, if it had to be replicated in ”bare metal” or physically. Between planning which hardware to buy, having to financially commit from the beginning and overdimensioning the needs of it thinking about traffic peaks, configuring everything, testing and resolving errors, etc, it would be more than one month before the hosting would be 100% operative. And after that, hardware maintenance costs, maintenance salaries and a large etcetera should be taken into account.

Economically speaking, in the 1.3.2 section I predicted that no more than $50 should be spent in tests during this months. At the end I have spent a total of $42.56. Initially I was using the AWS free tier but the more the infrastructure grew, the more the costs started adding up.

Month	Spendings
April	$4.82
May	$21.53
June	$16.21
TOTAL	$42.56

Table 7.1: Testing costs

7.2 Changes from initial purpose

There have been some changes from the initial idea I had. Some because it supposed doing too much work and it was not feasable and others because it was the logical thing to do.

- I do not intend to use Github Actions nor Ansible anymore. At first, I also wanted to do a CI/CD pipeline for deploying software into the created infrastructure, but unfortunately there was no time.
- I do not create a local database for storing the user-defined variables, instead I use a GitHub repository that stores them along with all the other source code.
- I do not use Qt or Java for programming the Guest User interface, I use Python with the pySimpleGUI library.
- Initially I wanted the platform to create, destroy and maintain the projects. However, at the end it will only create the service. It did not make sense to use the GUI for maintenance or destruction as this is done by a DevOps team working with version control systems.

7.3 Personal point of view

After developing this platform I could not believe how the previous way of building infrastructure was financially feasible. Lots of resources had to be dedicated to a developing phase that was not efficient at all. The flexibility, availability and low costs that not only the AWS cloud, but all the public clouds provide is incomparable.

Fast-paced DevOps environments require a way of setting up infrastructure that enables and enhances collaborative development and version control systems like GitHub provide exactly that.

Now that I am immersed in the DevOps and cloud worlds I know and can assure this is only the beginning. We can not forget that AWS was founded in 2006 and Terraform was released in mid 2014.It is one of the newest industries and lots of progresses and new technologies are yet to be made.

7.4 Future work

Although the platform is complete and works like a charm, improvements can always be made.

The infrastructure can add the AWS Shield Standard service to defend against DDoS attacks at no additional charge, because the infrastructure is already using Amazon CloudFront. This will improve the security aspect.

Serverless architecture is not explored at all in this thesis, but many experts say that improves efficiency and uses significantly less resources, but is much more complex to develop.

This project only focuses on the first part of the deployment of a web page. After the infrastructure is set, there are ways of automating software development releases. An example could be a CI/CD pipeline using Jenkins.

7.5 Technical competences consolidation

CTI1.1: Demonstrate an understanding of an organization's environment and its needs in the field of information and communication technologies. [Quite]

I built the platform thinking about the IT industry needs. Enhance the team work in a fast-paced environment is one of its key aspects.

CTI1.4: Select, design, deploy, integrate, evaluate, build, manage, operate and maintain hardware, software and networking technologies, within appropriate cost and quality parameters. [In depth]

The platform created automatically plans, builds and deploys a complex infrastructure, at lower costs than the classic approaches and higher quality.

CTI2.1: Direct, plan and coordinate the management of the computer infrastructure: hardware, software, networks and communications. [In depth]

I carefully planned the design of the infrastructure taking into account the management of hardware resources, software resources, networks and communications.

CTI3.3: Design, implement and configure networks and services. [Quite]

I designed and implemented more than 70 resources using more than 50 AWS services, including DNS records, private and public subnets, routing tables, internet gateways, NAT gateways, etc.

Acronyms list

ACL	Access Control List
ACM	Amazon Certificate Manager (AWS Service)
ALB	Application Load Balancer (AWS Service)
AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
AZ	Availability Zone
CCP	Certified Cloud Practitioner (AWS Certification)
CDN	Content Delivery Network
CI/CD	Continuous Integration / Continuous Deployment
CIDR	Classles Inter-Domain Routing
CMS	Content Management System
CPU	Central Processing Unit
DAFO	(In spanish) Debilidades, Amenazas, Fortalezas, Oportunidades
DNS	Domain Name System
EBS	Elastic Block Storage (AWS Service)
EC2	Elastic Compute Cloud (AWS Service)
EEE	Empresa i Entorn Econòmic (FIB subject)
EFS	Elastic File System (AWS Service)
EIP	Elastic IP address
GUI	Graphical User Interface
HA	High Availability
HCL	Hashicorp Configuration Language
IaC	Infrastructure as Code
IAM	Identity and Access Management (AWS Service)
IDE	Integrated Development Environment
IGW	Internet Gateway (AWS Service)
IP	Internet Protocol
IT	Information Technology
MI	Marketing a Internet (FIB subject)
NAT	Network Address Translation
RAM	Random Access Memory
RDS	Relational Database Service (AWS Service)
S3	Simple Storage Service (AWS Service)
SSL	Secure Socket Layer
TLD	Top Level Domain

VPC	Virtual Private Cloud (AWS Service)
WAF	Web Application Firewall (AWS Service)

Bibliography

[1] *Amazon Lightsail*. URL: https://aws.amazon.com/lightsail/. accessed: 02-2022.

[2] *Aurora vs. RDS: An Engineer's Guide to Choosing a Database*. URL: https://www.lastweekinaws.com/blog/aurora-vs-rds-an-engineers-guide-to-choosing-a-database/. accessed: 05-2022.

[3] *AWS*. URL: https://aws.amazon.com/. accessed: 02-2022.

[4] *AWS Certified Cloud Practitioner*. URL: https://aws.amazon.com/es/certification/certified-cloud-practitioner/. accessed: 02-2022.

[5] *AWS Global Infrastructure*. URL: https://aws.amazon.com/es/about-aws/global-infrastructure/. accessed: 03-2022.

[6] *AWS Managed Rules for AWS WAF*. URL: https://docs.aws.amazon.com/waf/latest/developerguide/aws-managed-rule-groups.html. accessed: 05-2022.

[7] *AWS RDS Aurora Terraform module*. URL: https://registry.terraform.io/modules/terraform-aws-modules/rds-aurora/aws/latest. accessed: 05-2022.

[8] *Configure S3 bucket as Terraform backend [Step-by-Step]*. URL: https://www.golinuxcloud.com/configure-s3-bucket-as-terraform-backend/#Step_1_Create_AWS_S3_bucket. accessed: 06-2022.

[9] *Create a secure and H/A VPC on AWS with Terraform*. URL: https://medium.com/muhammet-arslan/create-a-secure-and-h-a-vpc-on-aws-with-terraform-71b9b0a61151. accessed: 04-2022.

[10] *El proyecto EDINSOST*. URL: https://www.unicef.es/noticia/5-diferencias-entre-los-objetivos-de-desarrollo-del-milenio-y-los-. accessed: 02-2022.

[11] *GitHub*. URL: https://github.com/. accessed: 02-2022.

[12] *GitHub Repository with the code*. URL: https://github.com/j1nc0/TFG. accessed: 06-2022.

[13] *Glassdoor*. URL: https://www.glassdoor.es/index.htm?countryRedirect=true. accessed: 03-2022.

[14] *HashiCorp Cloud Engineer Certification - Terraform Associate*. URL: https://www.hashicorp.com/certification/terraform-associate. accessed: 02-2022.

[15] *HOW TO SECURE AWS TERRAFORM CREDENTIALS*. URL: https://letslearndevops.com/2017/07/24/how-to-secure-terraform-credentials/. accessed: 05-2022.

[16] *Infrastructure as code*. URL: https://en.wikipedia.org/wiki/Infrastructure_as_code. accessed: 02-2022.

[17] *Managing file system network accessibility*. URL: https://docs.aws.amazon.com/efs/latest/ug/manage-fs-access.html. accessed: 05-2022.

[18] *Protecting your ALB with WAF & Cloudfront*. URL: https://coresolutions.ltd/blog/protecting-your-alb-with-waf-cloudfront/. accessed: 05-2022.

[19] *Redis (cluster mode enabled vs disabled) vs Memcached*. URL: https://tutorialsdojo.com/redis-cluster-mode-enabled-vs-disabled-vs-memcached/. accessed: 04-2022.

[20] *Replication: Redis (Cluster Mode Disabled) vs. Redis (Cluster Mode Enabled)*. URL: https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/Replication.Redis-RedisCluster.html. accessed: 05-2022.

[21] *Should I use Memcached or Redis for WordPress caching?* URL: https://blog.kernl.us/2020/02/should-i-use-memcached-or-redis-for-wordpress-caching/. accessed: 05-2022.

[22] *Terraform*. URL: https://www.terraform.io/. accessed: 02-2022.

[23] *Terraform Language Documentation*. URL: https://www.terraform.io/language. accessed: 02-2022.

[24] *Trello*. URL: https://trello.com/. accessed: 02-2022.

[25] *WAFv2 Terraform Module*. URL: https://registry.terraform.io/modules/coresolutions-ltd/wafv2/aws/latest. accessed: 05-2022.

[26] *Web Application Hosting in the AWS Cloud*. URL: https://docs.aws.amazon.com/whitepapers/latest/web-application-hosting-best-practices/web-application-hosting-in-the-cloud-using-aws.html. accessed: 04-2022.

[27] *What is CI/CD?* URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd. accessed: 02-2022.

[28] *Wordpress Loadbalancing & Autoscaling on AWS using Elastic File System (EFS)*. URL: https://www.youtube.com/watch?v=bPN37KoAS3k&list=PLRCDFc1fbNkXTVS0zEJhl-WHxDdUnNNpd&index=19&t=3125s. accessed: 04-2022.