Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

Automatización de la gestión de la infraestructura en entorno cloud

Estudante: Jorge Berbel Carballal
Dirección: Alberto Caldas Lima
Dirección: Roberto Rey Expósito

A Coruña, febreiro de 2021.

A mi familia

Agradecimientos

A mi familia y amigos por apoyarme. A todas las nuevas amistades y compañeros con los que compartí grandes momentos durante estos años de universidad. A ambos directores por darme la oportunidad de poder realizar este TFG y a todos los compañeros de Everis que me han ayudado a llegar hasta aquí.

Resumen

El desarrollo de proyectos software necesita de diferentes entornos en los que ser probados, y cuando el volumen del proyecto es elevado, se hace necesario centralizar su gestión. Este Trabajo de Fin de Grado (TFG) propone la creación de un único mecanismo de gestión a través de un Command Line Interface (CLI), desarrollado en lenguaje Go, que facilite y centralice todas las operaciones que se pueden realizar en la infraestructura, la cual será desplegada en el proveedor cloud Amazon Web Services (AWS) mediante el uso del paradigma de Infraestructure as Code (IaC).

Además, con el objetivo de mantener un control y seguimiento de estos entornos, se desarrollarán mecanismos de estandarización de las instancias generadas, así como controles centralizados mediante una plataforma de monitorización global.

Abstract

The development of software projects needs different environments in which to be tested, and when the volume of the project is high, it´s necessary to centralize its management. This BSc Thesis proposes the creation of a single management mechanism through a Command Line Interface (CLI), developed in Go language, which facilitates and centralizes all the operations that can be carried out in the infrastructure, which will be deployed on the Amazon Web Services (AWS) cloud provider through the use of the Infrastructure as Code (IaC) paradigm.

Furthermore, with the aim of maintaining control and tracking of these environments, standardization mechanisms will be developed for the generated instances, as well as centralized controls through a global monitoring platform.

Palabras clave:

- CLI
- AWS
- IaC
- Terraform
- Packer
- Ansible
- Go
- Automatización
- Monitorización

Keywords:

- CLI
- AWS
- IaC
- Terraform
- Packer
- Ansible
- Go
- Automation
- Monitoring

Índice general

Índice de figuras

Índice de tablas

Índice de Listings

Capítulo 1

Introducción

En este primer capítulo de la memoria se expone una breve descripción de los motivos que han hecho que se lleve a cabo el presente Trabajo Fin de Grado (TFG), así como los principales objetivos que se desean lograr y la estructura en capítulos escogida para este documento. También se realiza una breve aclaración sobre el origen de este proyecto.

1.1 Motivación

Un proyecto de desarrollo de software transcurre por diferentes entornos de trabajo en los que ser desplegado y probado a lo largo de su ciclo de vida de desarrollo, y cuando el volumen del proyecto es elevado, se hace necesaria la centralización de su gestión. Podemos definir un entorno [1] como los recursos de infraestructura necesarios para acometer las tareas específicas requeridas por el producto software en función del estado en el que se encuentre. Habitualmente se suelen utilizar tres entornos: desarrollo, preproducción y producción. El entorno de desarrollo se utiliza desde el inicio del ciclo de vida del producto hasta la obtención de una versión mínimamente funciona y estable. En el siguiente entorno, el de preproducción, se llevan a cabo las pruebas finales del software desarrollado antes de su paso al entorno final. Como último entorno tenemos el de producción, donde tendríamos la aplicación en el estado disponible para el usuario final.

En la actualidad, la infraestructura que comprende a los entornos de trabajo descritos se suele desplegar a través de un proveedor en la nube debido a su escalabilidad, fiabilidad y al factor económico. Esta infraestructura se puede crear a partir de las herramientas que nos proporciona el propio proveedor cloud, que normalmente suelen ser un Command Line Interface (CLI) o una aplicación web, o bien utilizando alguna herramienta que implemente el paradigma de Infraestructure as Code (IaC) [2], que propone un aprovisionamiento completamente automatizado. Además, una vez que se ha desplegado una infraestructura surge la necesidad de comprobar que todo funciona correctamente y llevar un registro de lo que está

pasando, preferiblemente en tiempo real. Esto se obtiene mediante la observabilidad, que es la capacidad de inferir en los estados internos de un sistema, y se logra principalmente gracias a la monitorización y el uso de logs.

En este TFG se plantea la creación de un mecanismo de gestión único de proyectos a través de un CLI que facilite y centralice todas las operaciones que se pueden realizar sobre la infraestructura de entornos creada en un proveedor cloud. Además, con el objetivo de mantener un control y monitorización de estos entornos, se desarrollarán mecanismos de estandarización de las instancias creadas en la nube, así como controles centralizados empleando una plataforma de monitorización global.

1.2 Objetivos

Como se ha comentado en la sección anterior, el objetivo principal es desarrollar un CLI, en lenguaje Go [3], para gestionar los recursos de infraestructura de los proyectos, sobre todo cuando el volumen es elevado. Esta infraestructura será creada a través del proveedor cloud Amazon Web Services (AWS) [4] utilizando el paradigma de IaC, que es el proceso de aprovisionar y administrar la infraestructura en uno o más archivos de configuración en lugar de hacerlo manualmente mediante una interfaz de usuario. El propio proveedor AWS ofrece el servicio CloudFormation para realizar esta tarea, pero se ha optado por utilizar la herramienta Terraform [5] y conseguir de este modo una implementación que soporte entornos multicloud. Para crear y configurar las plantillas que usarán las instancias de la infraestructura se utilizará la herramienta Packer [6]. Por último, para el proceso de monitorización de datos se recurrirá a Prometheus [7] y a Grafana [8] para su visualización. Con esto lo que se desea lograr es poder facilitar y centralizar las operaciones de los proyectos y mantener un control de los entornos. A continuación se listan los objetivos concretos que se han de alcanzar, siguiendo el orden proporcionado, para finalmente llegar a cumplir el objetivo final del TFG. Estos objetivos son:

- Diseño de la infraestructura para dar soporte a diferentes entornos y aplicaciones en un entorno cloud.
- Definición de la creación de los componentes de la infraestructura siguiendo el paradigma IaC.
- Configuración de las instancias base de las máquinas que se utilizarán en esos entornos.
- Diseño y configuración de una plataforma de monitorización centralizada de la infraestructura.
- Desarrollo de un CLI para la gestión de los recursos de infraestructura de los proyectos.

1.3 Origen del proyecto

Cabe resaltar que este TFG no parte completamente de cero, si no que se ha continuado tanto con el desarrollo de la infraestructura como con el del CLI a partir de una iteración inicial. Los diseños e implementaciones finalmente alcanzados distan bastante de la primera aproximación, en los que se han añadido y eliminado numerosos elementos. En los Capítulos 4 y 5 del presente documento se explican estos aspectos donde se entrará más en detalle a exponer el punto de partida y qué se ha modificado exactamente.

1.4 Estructura de la memoria

Seguidamente se muestra la estructuración y una breve descripción del contenido de los capítulos del presente documento:

1. **Introducción**: Primer y actual capítulo donde se exponen los motivos de la realización de este TFG, así como sus objetivos y de donde parte, para finalizar con la estructura en capítulos del documento.

2. **Tecnologías y herramientas**: Segundo capítulo donde se realiza una breve explicación sobre las tecnologías y herramientas utilizadas a lo largo del TFG.

3. **Metodología**: Tercer capítulo de la memoria, en el cual se describe la metodología de desarrollo ágil Scrum y su aplicación al TFG.

4. **Diseño e implementación de la infraestructura**: En este cuarto capítulo se explica la infraestructura elaborada en AWS, comenzando por su diseño y continuando con su implementación basándose en el paradigma IaC. También se trata el tema de la monitorización de los entornos con la herramienta Prometheus.

5. **Diseño e implementación del CLI**: Quinto capítulo que sigue un patrón parecido al anterior. Primero se realizará el diseño del CLI en lenguaje Go y posteriormente su implementación. También se dedica una sección para explicar la estimación de tiempo y coste del desarrollo completo de proyecto y otra sección donde se explican las pruebas realizadas.

6. **Conclusiones y trabajo futuro**: Sexto y último capítulo de la memoria donde se enumeran las principales conclusiones obtenidas y posibles líneas de trabajo futuro.

Capítulo 2

Tecnologías y herramientas

En este capítulo se introduce al lector en las tecnologías y herramientas utilizadas a lo largo del proyecto. Además, se explicarán varios conceptos básicos que sirven de introducción al funcionamiento de alguna de ellas.

2.1 Amazon Web Services

Amazon Web Services (AWS) [9] es una plataforma de computación en la nube proporcionada por Amazon que ofrece una gran variedad de servicios, tales como cómputo, almacenamiento, bases de datos, análisis o redes, entre muchos otros. Para dar estos servicios AWS tiene centros de datos distribuidos por todo el mundo. Cualquier servicio que deseemos utilizar tiene que estar asociado a uno de estos centro de datos. AWS denomina región [10] a cada una de las diferentes zonas geográficas donde se agrupan. Cada región consta de varias zonas de disponibilidad (Availability Zones (AZ)) que son uno o más centros de datos discretos que existen dentro de cada región con alimentación, refrigeración, redes y conectividad redundantes e independientes entre ellos. De esta forma, AWS ofrece alta disponibilidad, tolerancia a fallos y escalabilidad en todos sus servicios.

A continuación se introducen algunos de los servicios básicos de AWS que serán utilizados a la hora de implementar la infraestructura o el CLI:

- Amazon Elastic Compute Cloud (EC2) [11] es un servicio web que proporciona capacidad de computación escalable en la nube. Esta capacidad es utilizada a través de entornos informáticos virtuales denominados instancias. Una instancia se crea a partir de una Amazon Machine Image (AMI), que es una plantilla que contiene una configuración específica de software (sistema operativo, servidor de aplicaciones...), y que utiliza un servicio de almacenamiento basado en volúmenes de alto rendimiento conocido como Amazon Elastic Block Store (EBS). Este servicio EBS se puede entender como dispositivos de bloques (discos) que se pueden adjuntar a una instancia al inicializarla o cuando

está en ejecución. Una instancia se puede encontrar en uno de los siguientes tres estados: en ejecución (running), parada (stop) o terminada (terminate). La conexión segura con una instancia EC2 se realiza habitualmente configurando un par de claves, pública y privada, de manera que la instancia almacena la clave pública y el usuario accede a ella usando la clave privada.

- Amazon S3 (Amazon Simple Storage Service) [12] es un servicio de almacenamiento de objetos escalable en la nube. Entendemos como objeto a las entidades fundamentales almacenadas en Amazon S3, que se componen de datos de objeto y metadatos. Presenta un interfaz de servicios web simple para almacenar y recuperar cualquier cantidad de datos, en cualquier lugar y parte de la web. Los objetos son almacenados en contenedores llamados buckets.

- Amazon Virtual Private Cloud (VPC) [13] es la capa de red para Amazon EC2, que permite lanzar recursos de AWS en una red virtual. Esta red virtual es prácticamente idéntica a las redes tradicionales y está aislada lógicamente de otras VPCs. Algunas de las funcionalidades que permiten las VPCs son: añadir subredes, configurar tablas de enrutamiento o asociar grupos de seguridad (firewalls virtuales a nivel de instancia).

- Amazon DynamoDB [14] es un servicio de base de datos NoSQL muy flexible.

- Amazon (Identity and Access Management (IAM) [15] es el servicio encargado de la creación y administración de usuarios para AWS, así como de otorgar los correspondientes permisos para que dichos usuarios obtengan acceso a los recursos.

2.2 Terraform

Terraform [16] es una herramienta de automatización IaC desarrollada por HashiCorp que permite crear, cambiar y versionar infraestructura de manera segura y eficiente. Utiliza un lenguaje de configuración propio denominado HashiCorp Configuration Language (HCL). Terraform utiliza HCL para crear archivos de configuración, con extensión .tf, que contienen definiciones de los recursos deseados soportando casi cualquier proveedor (AWS, Azure, Docker...), automatizando la creación de dichos recursos. Además, Terraform es una plataforma multicloud agnóstica, es decir, se puede gestionar un entorno heterogéneo con el mismo flujo de trabajo. Este flujo de trabajo (véase Figura 2.1) es relativamente sencillo y se resume en dos pasos principales:

1. Se inicializa el directorio donde se encuentran los archivos de configuración HCL y se genera un fichero de estado. Este fichero de estado será utilizado en local para mantener un registro de la infraestructura creada. En el caso de que se realice un cambio

o se agregue un recurso a una configuración determinada, Terraform comparará esas modificaciones con el fichero de estado para determinar qué cambios resultan en nuevo recursos o en modificaciones de recursos existentes.

2. Terraform genera un plan de ejecución con los archivos de configuración del directorio previamente inicializado. Este plan describe lo que hará para alcanzar el estado deseado y luego lo ejecuta para construir la infraestructura descrita.

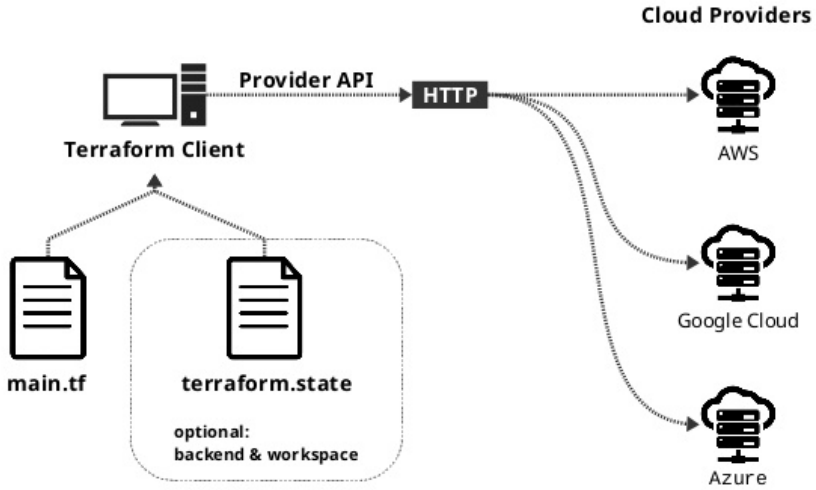


Figura 2.1: Flujo de trabajo de Terraform

Las operaciones descritas se corresponden con los comandos *terraform init*, para inicializar, y *terraform apply*, para generar el plan de ejecución y la creación de los recursos, así como la escritura en el fichero de estado. También existe el comando *terraform destroy*, opuesto a *terraform apply*, que sirve para destruir la infraestructura creada.

Dentro de los archivos de configuración se distinguen tres tipos de bloques:

- Terraform block: Es donde se define el proveedor que se va a utilizar para que Terraform pueda descargar las bibliotecas correspondientes desde el Terraform Registry.
- Providers block: Son plugins que Terraform usa para gestionar los recursos. Se encargan de traducir las interacciones de la API con el servicio. Cualquier servicio soportado tiene un provider que define los recursos que están disponibles.
- Resources block: Cada bloque de resource describe uno o varios objetos de la infraestructura. Los resources que se utilizan conjuntamente se suelen agrupar en módulos.

Finalmente, en la Figura 2.2 se muestra un diagrama de la estructura de Terraform.

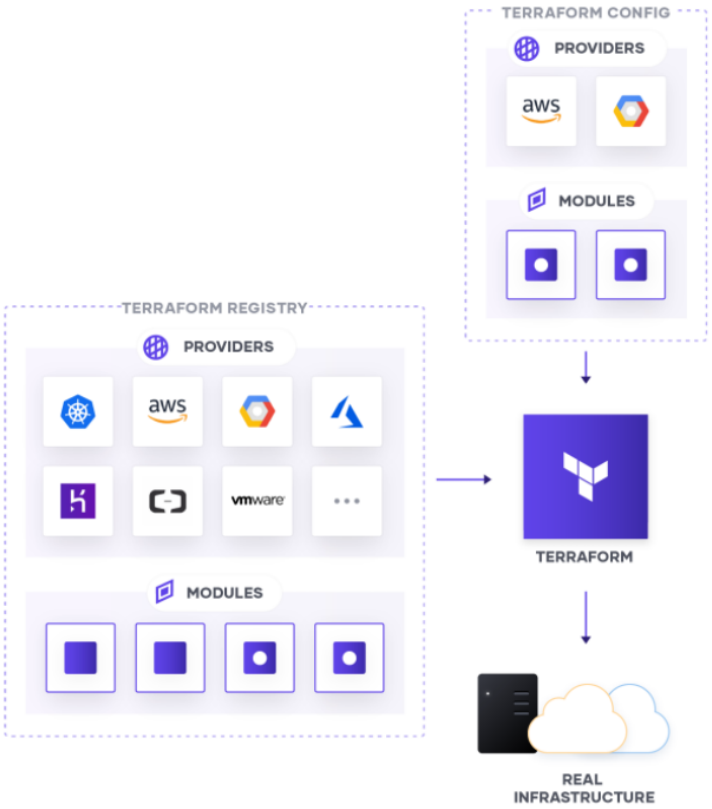


Figura 2.2: Estructura de Terraform

2.3 Packer

Packer [17] es una herramienta IaC de código abierto también desarrollada por HashiCorp para crear imágenes máquina idénticas para múltiples plataformas a partir de una configuración fuente única. Se entiende como imagen máquina una unidad estática que contiene un sistema operativo preconfigurado y un software preinstalado.

Algunos ejemplos de uso de Packer son la creación de AMIs personalizadas para Amazon EC2 u Open Virtualization Format (OVF) para VirtualBox. El archivo de configuración donde se define la imagen que se desea construir se denomina template y su formato es JSON. Como componentes principales dentro de un template distinguimos entre:

- Builder: Es el componente de Packer encargado de crear la imagen máquina y convertirla en un formato válido para una o varias plataformas.

- Provisioners: Permiten configurar e instalar software en la imagen máquina después del arranque.

2.4 Ansible

Ansible [18] es una herramienta de automatización TI. Puede automatizar el aprovisionamiento de software, configurar sistemas, desplegar aplicaciones y orquestar tareas TI más avanzadas, como implementaciones continuas o actualizaciones continuas sin tiempo de inactividad, todo ello de una forma sencilla, robusta y paralela. Es una herramienta de software libre de carácter declarativo, compatible con cualquier sistema UNIX y es *agentless*, es decir, no necesita un agente en la máquina donde se ejecuta la acción. La arquitectura de Ansible, mostrada en la Figura 2.3, está formada por dos componentes principales: el nodo controlador (Control/Management Node) y el/los nodo(s) gestionado(s) (Managed Node(s)).

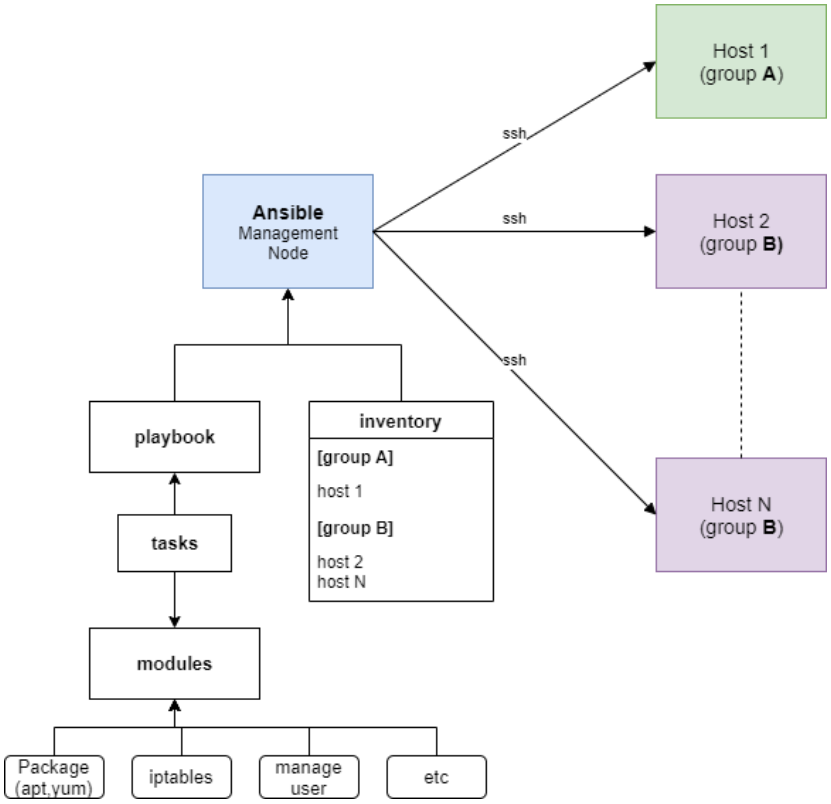


Figura 2.3: Arquitectura general de Ansible

Las funcionalidades que ofrece esta herramienta son realizadas a través de tareas (tasks), que son las responsables de ejecutar las bibliotecas de Ansible, denominadas módulos. Cada módulo tiene un uso y objetivo particular y se pueden agrupar en colecciones. El formato utilizado para definir las tareas es lenguaje YAML, y el archivo donde se define una o más tareas se denomina playbook. En un playbook, las tareas se ejecutan de una en una y en orden secuencial. A modo de ejemplo, el fragmento de código 2.1 muestra la sintaxis general de un playbook.

```
---
- name: Nombre del playbook
  hosts: hosts sobre los que se aplica
  become: yes/no (para dar privilegios de superusuario)
  vars: (declaración de posibles variables)
  tasks: (ejemplo de una tarea)
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest
```

Listing 2.1: Ejemplo de sintaxis de un playbook de Ansible

El contenido de los playbooks se suele agrupar en los llamados roles de Ansible. Se basan en la idea de incluir archivos y combinarlos para formar abstracciones reutilizables, lo que permite crear un playbook con una mínima configuración y definir toda la complejidad y lógica de las acciones a más bajo nivel. Para ello, hay que crear una estructura de directorios y ficheros manualmente o a partir de un repositorio de roles, conocido como Ansible Galaxy [19], donde los usuarios pueden subir sus propios roles para dar soporte a un mayor número de funcionalidades.

El funcionamiento general es simple: Ansible se instala únicamente en una máquina (el nodo controlador) que usará el protocolo SSH (por defecto) para comunicarse con las máquinas que se desea gestionar (los nodos gestionados o hosts), los cuales no necesitan disponer de Ansible instalado. Para que esto sea posible, ambos tipos de nodos necesitan tener instalado Python y SSH. Para ejecutar Ansible se debe crear un archivo denominado inventario en el nodo controlador, que almacena una lista de las direcciones IP o nombres de máquina de los hosts a gestionar, entre otro tipo información. Después se ejecuta el playbook donde se han definido las tareas que se desean realizar en los nodos listados en el inventario. Por último, los módulos de Ansible, asociados a las tareas del playbook, se copian en los nodos gestionados automáticamente y se ejecuta su función.

2.5 Prometheus

Prometheus [20] es una herramienta de monitorización y alerta de sistemas de código abierto. Nace a partir de Borgmon, una herramienta interna de Google con las mismas características. Prometheus fundamentalmente almacena los datos como series de tiempo, que son secuencias de valores con marca de tiempo que pertenecen a las misma métrica y al mismo conjunto de dimensiones etiquetadas. Cada serie de tiempo se identifica de forma única por su nombre de métrica y pares clave-valor.

Prometheus proporciona un lenguaje de consulta funcional llamado Prometheus Query Language (PromQL) que permite al usuario seleccionar y agregar datos de series de tiempo en tiempo real. La recopilación de series de tiempo ocurre a través de un modelo de extracción

sobre protocolo HTTP. En la Figura 2.4 se muestra un diagrama general de la arquitectura de Prometheus.

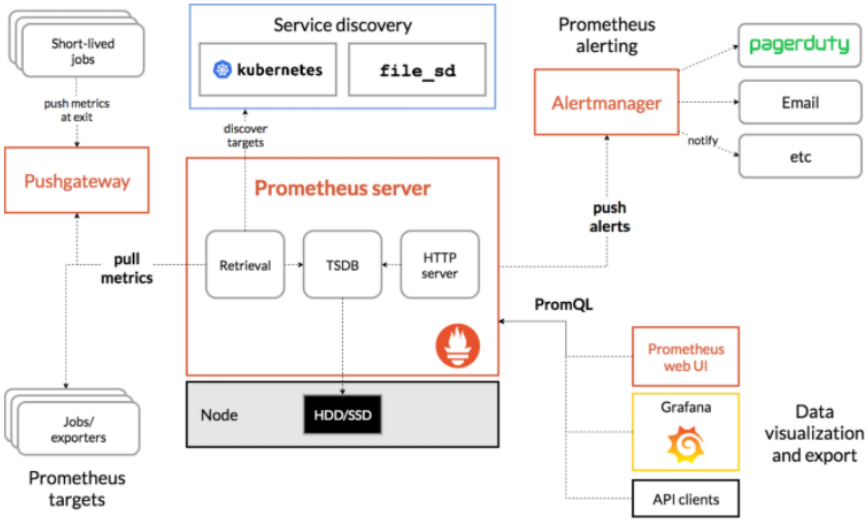


Figura 2.4: Arquitectura Prometheus

Como se pueda ver en la figura, Prometheus extrae métricas de los denominados prometheus targets, que son los endpoints a partir de los cuales se obtienen las métricas. Estos endpoints se definen como instancias según la terminología propia de la herramienta y a un conjunto de instancias se le denomina job. El prometheus server extrae y almacena los datos de series de tiempo y ejecuta reglas sobre esos datos para producir nuevas series de tiempo o generar alertas (AlertManager). Para visualizar los datos recopilados se utilizan consultas PromQL con algún API como puede ser la plataforma de observabilidad Grafana.

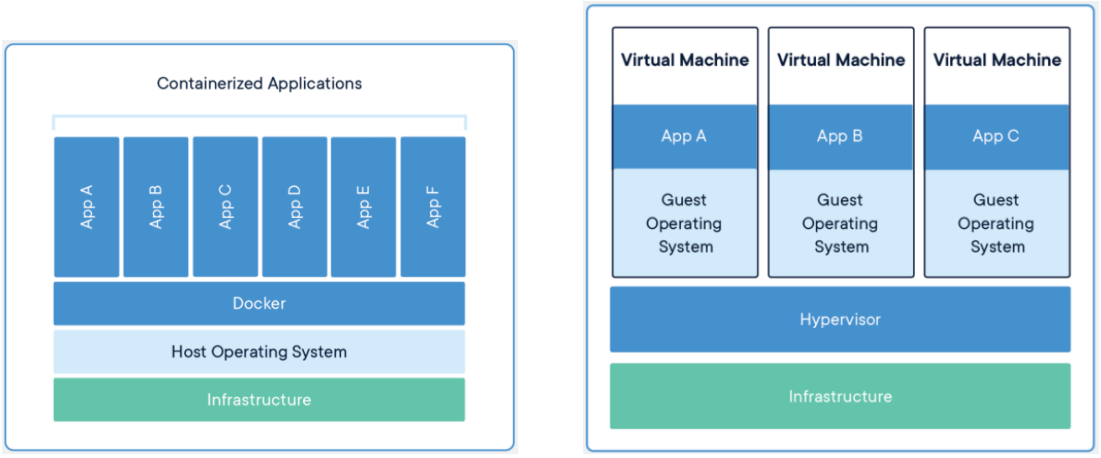
2.6 Grafana

Grafana [21] es una herramienta de análisis y visualización de datos de series temporales. Es de código abierto y permite consultar, visualizar y alertar sobre métricas a través de dashboards o cuadros de mando. Grafana soporta múltiples fuentes de datos, como por ejemplo Prometheus, que además se pueden mezclar en un mismo dashboard.

2.7 Docker

Docker [22] es una plataforma de software para desarrollar, implementar y probar aplicaciones, permitiendo aislarlas de la infraestructura. Docker proporciona la habilidad de empaquetar y ejecutar las aplicaciones en un entorno estandarizado y aislado llamado contenedor.

Un contenedor (véase Figura 2.5a) empaqueta el código y todas sus dependencias para que la aplicación se ejecute de forma rápida y segura en diferentes entornos informáticos. El software de un contenedor siempre se ejecutará de la misma manera, independientemente de la infraestructura. Los contenedores son diferentes a las máquinas virtuales (VM) ya que se ejecutan directamente sobre el kernel de la máquina host. Una máquina virtual (véase Figura 2.5b) es un entorno que funciona como un sistema informático virtual con su propia CPU, memoria, interfaz de red y almacenamiento pero son recursos virtuales que surgen de una abstracción del hardware físico [23]. El hipervisor es la capa software encargada de realizar la virtualización, es decir, separar los recursos hardware de la máquina del sistema e implementarlos para que la VM pueda utilizarlos.



(a) Docker Container

(b) Máquina virtual

Figura 2.5: Comparativa entre un contenedor Docker y una máquina virtual

Docker utiliza un arquitectura cliente-servidor. El cliente se comunica con el demonio de Docker que se encarga de compilar, ejecutar y distribuir los contenedores. La comunicación se produce a través de una API REST y ambos pueden ejecutarse en el mismo sistema o se puede conectar un cliente con un demonio remoto. Los contenedores Docker se crean a partir de una plantilla denominada imagen. Las imágenes de Docker se pueden almacenar en el Docker Registry, que se encuentra del lado del servidor. La Figura 2.6 muestra un diagrama de la arquitectura de Docker.

El funcionamiento básico de Docker es el siguiente: primero se debe crear un fichero Dockerfile, que es como un script de instrucciones que se utiliza para crear una imagen de un contenedor, que se convertirá en una sola aplicación. Luego, con el comando *docker build* construiremos una imagen a partir del Dockerfile y con el comando *docker run* crearemos el contenedor a partir de la imagen. Para facilitar la creación de varios contenedores con múltiples servicios disponemos de Docker Compose, que se desarrolló para ayudar a definir y

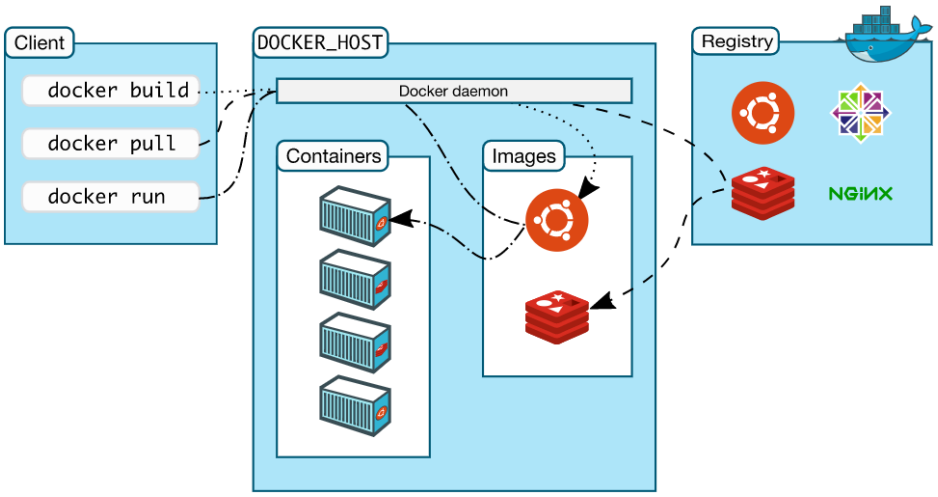


Figura 2.6: Arquitectura general de Docker

compartir servicios para varios contenedores. Se utiliza un archivo en formato YAML para configurar los diferentes contenedores (servicios) que componen la aplicación.

2.8 VirtualBox

Oracle VM VirtualBox [24] es un software de virtualización multiplataforma para hardware x86 y x86-64 que permite a los usuarios ejecutar múltiples sistemas operativos al mismo tiempo en forma de máquinas virtuales. Se puede ejecutar desde línea de comandos o desde Graphical User Interfac (GUI). Un solo host de Virtualbox puede implementar tantas máquinas virtuales como su hardware permita, teniendo en cuenta que siempre debe dejar recursos suficientes para que el propio host funcione correctamente. Puede ser instalado en Microsoft Windows, macOS, Linux y Oracle Solaris, mientras que la máquina virtual que se cree puede utilizar cualquier distribución Linux, Solaris, macOS, BSD, IBM OS/2 o Windows.

2.9 Go

Go [25, 26] es un lenguaje de programación de código abierto que facilita la creación de software simple, confiable y eficiente. Fue creado en 2007 por Robert Griesemer, Rob Pike y Ken Thompson, desarrolladores de Google, y anunciado en 2009. Funciona en Unix, FreeBSD, OpenBSD, macOS, Plan 9 y en Microsoft Windows. Tiene gran similitudes con C pero también está basado en otros lenguajes de programación. El sistema de tipos que utiliza Go permite la construcción de programas flexibles y modulares. Sus mecanismos de concurrencia son novedosos y eficientes y el enfoque de la abstracción de datos y la programación orientada a

objetos son tremendamente flexibles. Además, posee gestión automática de memoria mediante un recolector de basura. Es un lenguaje compilado rápido, de tipado estático que se siente como un lenguaje interpretado de tipado dinámico.

2.10 Git

Git [27] es un sistema de control de versiones distribuido, gratuito y de código abierto diseñado para manejar todo tipo de proyectos, desde los más pequeños hasta los más grandes, con velocidad y eficiencia. También, permite la utilización de ramas o branches que son independientes entre ellas y se pueden crear, fusionar (merge) o borrar en cuestión de segundos. Actualmente es el sistema de control de versiones más utilizado en el mundo.

2.11 Visual Studio Code

Visual Studio Code [28] es un editor de código fuente desarrollado por Microsoft gratuito y de código abierto. Soporta casi cualquier lenguaje de programación existente y presenta un apartado denominado extensiones (plugins) que nos permitirá ampliar sus funcionalidades, permitiéndonos añadir un control integrado de Git o Docker. El propio editor tiene la funcionalidad de abrir un terminal, y en este TFG se realiza la compilación de los archivos tanto desde el CMD como del WSL.

2.12 Windows Subsystem for Linux

El subsistema de Windows para Linux o Windows Subsystem for Linux (WSL) [29] es una capa de compatibilidad desarrollada por Microsoft que permite ejecutar herramientas de línea de comandos nativas de Linux directamente en Windows 10 a través de un terminal, junto con sus aplicaciones tradicionales de escritorio. WSL requiere menos recursos (CPU, memoria y almacenamiento) que una máquina virtual, siendo una buena alternativa para los desarrolladores. También es posible acceder a los archivos de Windows desde Linux, lo que permite utilizar aplicaciones de Windows y herramientas de línea de comandos de Linux en el mismo conjunto de archivos.

2.13 MobaXterm

MobaXterm [30] es una herramienta de gestión de conexiones remotas para Windows. Proporciona todas las utilidades de gestión de conexiones de red remota más importantes como demonios de red básicos y administradores de sesión (SSH, FTP, RDP, Telnet, ...), y

un terminal multilab con comandos UNIX integrados (bash, ls, cat, rsync, ...) al escritorio de Windows. Además, presenta un constructor gráfico de túneles SSH.

Capítulo 3

Metodología

Como metodología de desarrollo para este proyecto se ha decidido utilizar el modelo de desarrollo ágil Scrum. Antes de explicar en que consiste esta metodología, tenemos que explicar qué es un modelo de desarrollo ágil. Los modelos de desarrollo ágil o métodos ágiles se centran en un desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan con el tiempo en función de las necesidades del proyecto. Estos modelos nacieron como alternativa a las metodologías ya existentes basándose en cuatro postulados recogidos dentro del denominado Manifiesto Ágil [31] elaborado en 2001, que se cita a continuación:

> «Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:
>
> - Individuos e interacciones sobre procesos y herramientas
> - Software funcionando sobre documentación excesiva
> - Colaboración con el cliente sobre negociación contractual
> - Respuesta ante el cambio sobre seguir un plan
>
> Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.»

3.1 Scrum

Scrum [32, 33] es un marco de trabajo para el desarrollo y mantenimiento de productos complejos caracterizado por:

- Llevar a cabo una gestión evolutiva del producto, en lugar de la tradicional o predictiva.
- Basar la calidad del resultado más en el conocimiento tácito de las personas en equipos autoorganizados, que en la calidad de los procesos empleados.

- Usar una estrategia de desarrollo incremental a través de iteraciones (Sprints).

Strum está compuesto por:

- Roles:
  - El Equipo de Desarrollo (Development Team).
  - El Propietario del Producto (Product Owner).
  - El Scrum Master.
  - Interesados (Stakeholders).
- Artefactos:
  - Pila de Producto (Product Backlog).
  - Pila de Sprint (Sprint Backlog).
  - Incremento.
- Eventos:
  - Sprint.
  - Reunión de Planificación del Sprint (Sprint Planning Meeting).
  - Scrum Diario (Daily Scrum).
  - Revisión del Sprint (Sprint Review).
  - Retrospectiva del Sprint (Sprint Retrospective).

En la Figura 3.1 podemos observar el ciclo de desarrollo del marco Scrum así como los elementos por los que está formado, que se explican a continuación. Antes de entrar en detalle con los componentes de Scrum debemos definir su pieza más importante: el Sprint. Se denomina Sprint a cada ciclo o iteración de trabajo que produce una parte del producto terminada y funcionalmente operativa, conocida como incremento.

3.1.1 Roles

Son todas las personas que intervienen o tienen relación directa o indirecta con el proyecto. Se distinguen tres roles:

- Propietario del Producto (Product Owner): El Product Owner es la persona que toma las decisiones del cliente. Su responsabilidad es lograr el máximo valor de producto para los clientes, usuarios y resto de implicados. Este rol debe ser llevado a cabo por una única persona para poder simplificar la comunicación y la toma de decisiones. También es la única persona responsable de gestionar el Product Backlog.

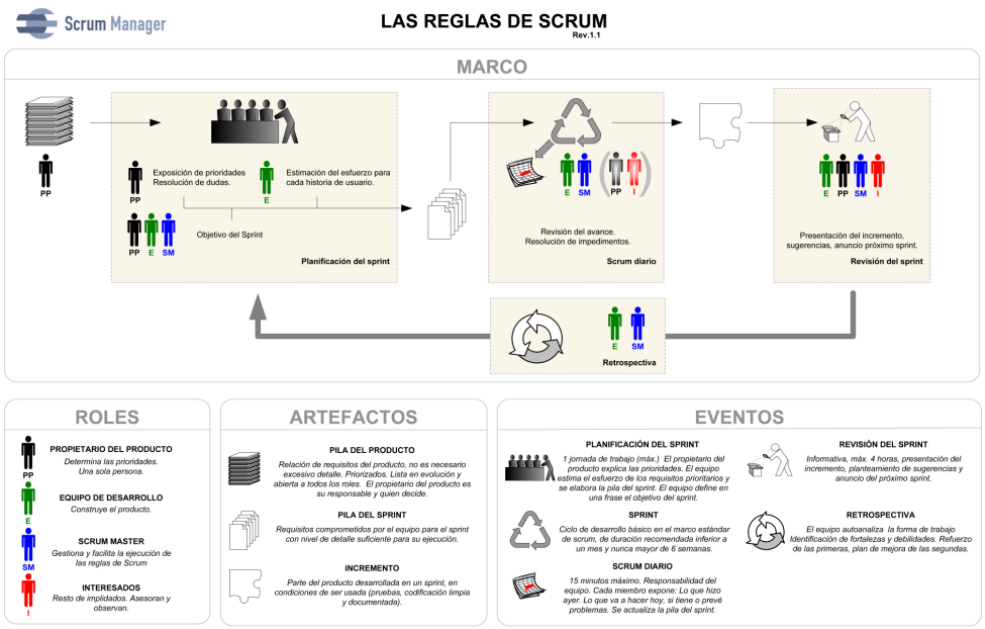


Figura 3.1: Marco Scrum

- Equipo de Desarrollo (Development Team): Lo forman el grupo de profesionales que realizan el incremento de cada Sprint. Es un equipo multifuncional y autoorganizado de entre tres y nueve personas. No hay un gestor para asignar y coordinar tareas, si no que son los propios miembros del equipo los que se tienen que poner de acuerdo para gestionar su propio trabajo.
- Scrum Master: Es el responsable de asegurar que el método Scrum es entendido y adoptado, proporcionando asesoría y formación necesaria al Propietario del Producto y al Equipo de Desarrollo. Además, se encarga de resolver posibles bloqueos del Equipo de Desarrollo.
- Interesados (Stakeholders): Resto de implicados que acuden a la revisión del Sprint para asesorar y observar. Ocasionalmente pueden acudir al Scrum diario con la misma motivación.

3.1.2 Artefactos

Los artefactos están diseñados para maximizar la transparencia de la información clave, que es necesaria para asegurar que todos tengan el mismo entendimiento del artefacto. Se distinguen los siguientes tipos:

- Pila de Producto (Product Backlog): Es una lista ordenada de todo aquello que el Product Owner cree que necesita el producto a desarrollar, y es la única fuente de requisitos

para cualquier cambio a realizarse en el mismo. La lista está compuesta por las funcionalidades, mejoras y corrección de errores que deben incorporarse al producto a través de cada Sprint. Es importante recalcar que la lista de producto nunca está completa, al principio del proyecto incluye los requisitos inicialmente conocidos y mejor entendidos, que evolucionan conforme avanza el desarrollo.

El formato de la pila suele constar habitualmente de tres partes: descripción de la funcionalidad (historia de usuario), prioridad y preestimación del esfuerzo necesario. Si la historia de usuario que describe la funcionalidad es de gran tamaño se le denomina épica, puesto que debe ser desglosada en tareas más pequeñas. El Product Owner mantiene la pila ordenada por la prioridad de los elementos, siendo los más prioritarios los que aportan mayor valor al producto.

- Pila de Sprint (Sprint Backlog): Es la lista de tareas, seleccionadas de la Pila de Producto, necesarias para construir los requisitos que se van a realizar en un Sprint. La crea el equipo de desarrollo en la reunión de planificación del Sprint, indicando el esfuerzo previsto para la realización de cada tarea.

  Las historias de usuario seleccionadas se descomponen en unidades de tamaño adecuadas para monitorizar el avance diario. Si por cualquier motivo no se pudieran completar alguna de las tareas actuales, deberán pasar a la siguiente Pila de Sprint.

- Incremento: Es la parte de producto realizada en un Sprint potencialmente entregable, es decir, terminado y probado.

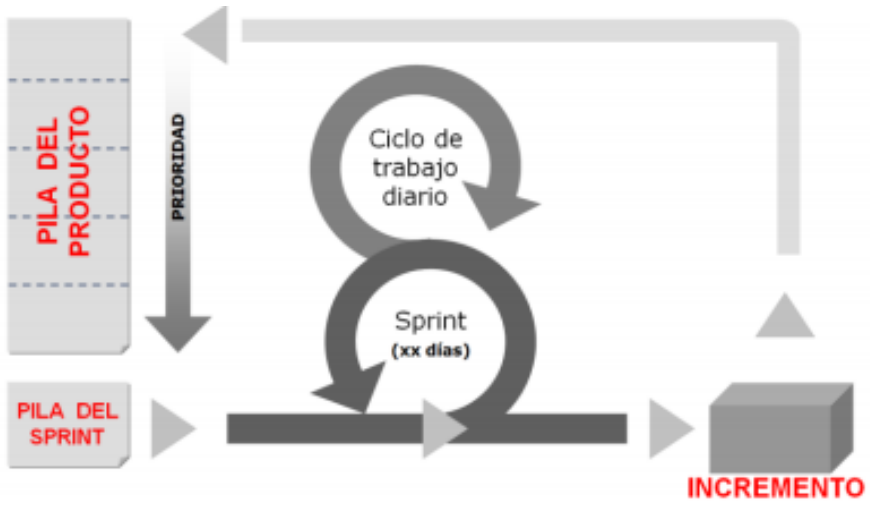


Figura 3.2: Artefactos Scrum

3.1.3 Eventos

Los eventos son bloques de tiempo (time-boxes) con una duración máxima.

- Sprint: Bloque de tiempo de máximo un mes donde se genera un incremento de producto. Su duración es fija, a diferencia del resto de eventos que pueden terminar si su objetivo ha sido cumplido.
- Reunión de Planificación del Sprint (Sprint Planning Meeting): Reunión de trabajo que marca el inicio de cada sprint, compuesta por todos los roles, en la que se determina cuál va a ser el objetivo del Sprint (Sprint Goal) y sus tareas para conseguirlo.
- Scrum Diario (Daily Scrum): Breve reunión diaria del equipo de desarrollo donde se tratan tres cuestiones:
  - Trabajo realizado el día anterior.
  - Trabajo previsto para el día de hoy.
  - Existencia de algún bloqueo que impida realizar el trabajo.

  Posteriormente cada persona debe actualizar en la Pila de Sprint la estimación de esfuerzo pendiente en sus tareas.
- Revisión del Sprint (Sprint Review): Al finalizar el Sprint se lleva a cabo una reunión para analizar el incremento generado y realizar una adaptación a la Pila del Producto si fuera necesario. Cabe destacar que se trata de una reunión informativa, donde la presentación del incremento tiene como propósito facilitar la retroalimentación de la información. Como resultado de esta reunión se obtiene la Pila del Producto revisada.
- Retrospectiva del Sprint (Sprint Retrospective): Es una reunión para la mejora del marco de trabajo, donde el equipo realiza un auto-análisis de su forma de trabajo, identificando fortalezas y puntos débiles.

3.2 Scrum aplicado al proyecto

El TFG se ha desarrollado en una empresa en el marco de unas prácticas extracurriculares, y en el que podemos diferenciar dos roles. Por un lado, el tutor profesional del TFG (Alberto), que desempeña las funciones de Product Owner y Scrum Master. Y por otro lado, el autor del TFG que realiza la función de Development Team. También aparece la figura del interesado, desempeñada por tres compañeros de la empresa que realizan la función de asesoramiento en las revisiones del Sprint.

Lo primero que se realiza es el Product Backlog inicial por parte del Product Owner, para fijar las tareas base del proyecto y cumplir con los objetivos fijados en inicio del TFG (véase Sección 1.2). Posteriormente se decide el número de Sprints necesarios y su duración para cumplir con la previsión de entrega. Se determinan 4 Sprints para llevar a cabo esas tareas y su duración dependerá de las tareas escogidas en cada caso, aunque siempre sin superar el tiempo máximo de 4 semanas.

A lo largo del desarrollo se realizan reuniones diarias entre el Development Team y el Product Owner para responder a las cuestiones comentadas en el apartado Scrum diario de la Sección 3.1.3. Antes de iniciar el desarrollo de las tareas de cada Sprint se lleva a cabo una reunión de planificación y después de finalizarlo se realiza una reunión de revisión del Sprint, donde el Equipo de Desarrollo efectúa una demostración del incremento obtenido. Además se revisa el Product Backlog por si fuera pertinente realizar alguna modificación. A lo largo del proyecto se ha optado por no realizar ninguna reunión de retrospectiva del Sprint.

A continuación se muestra el Product Backlog inicial en la Tabla 3.1 y la estimación de los Sprints del TFG en la Tabla 3.2, donde el esfuerzo se mide en puntos de dificultad y no en horas/hombre. Las metodologías ágiles basan la estimación del esfuerzo en el conocimiento, por ello, como el equipo de desarrollo del TFG está formado por una persona y, teniendo en cuenta la experiencia adquirida, se puede hacer una conversión entre puntos de esfuerzo y horas/hombre a razón de 2,5.

Los Sprints serán explicados en más detalle en los Capítulos 4 y 5 del presente documento. Para organizar el flujo de trabajo del proyecto se utiliza una estructuración en carpetas usando un repositorio Git.

Id	**Descripción de la funcionalidad**	**Prioridad**	**Estimación del esfuerzo**
1	Diseño de la infraestructura de entornos para AWS	Muy alta	4
2	Implementación de la infraestructura en AWS siguiendo el paradigma IaC	Muy alta	28
3	Diseño de sistema de monitorización de la infraestructura en AWS	Muy alta	4
4	Implementación de sistema de monitorización de la infraestructura en AWS	Muy alta	6
5	Diseño del CLI para realizar operaciones sobre la infraestructura en AWS	Muy alta	16
6	Implementación del CLI para poder crear comandos que realicen las operaciones deseadas sobre la infraestructura en AWS	Muy alta	8
7	Un usuario levanta una o varias instancias para un proyecto concreto en el entorno seleccionado de la infraestructura en AWS a través del CLI	Muy alta	8
8	Un usuario levanta una AMI propia en AWS a través del CLI	Alta	3

Tabla 3.1: Product Backlog inicial

Sprint	Id del Product Backlog	Comienzo	Fin	Esfuerzo estimado
Sprint 1	1, 2	5/10/2020	29/10/2020	32
Sprint 2	3, 4	2/11/2020	9/11/2020	10
Sprint 3	5	10/11/2020	23/11/2020	16
Sprint 4	6, 7, 8	24/11/2020	09/12/2020	19

Tabla 3.2: Estimación de los Sprints

Capítulo 4

Diseño e implementación de la infraestructura

En este capítulo se explica el diseño y la implementación de la infraestructura creada en el entorno cloud de AWS a partir de las tareas seleccionadas del Product Backlog (véase Tabla 3.1 en la Sección 3.2). Las secciones del presente capítulo se corresponden con los Sprints 1 y 2 mostrados en la Tabla 3.2, exceptuando la primera sección que se corresponde con una introducción a las herramientas usadas y una breve explicación del origen de la infraestructura. En cada sección se muestra una tabla detallada del Sprint correspondiente con todas sus tareas realizadas así como el esfuerzo y coste.

4.1 Preparación y estudio actual de la infraestructura

Antes de iniciar la parte de desarrollo del proyecto se dedicaron dos semanas a la instalación y configuración de todas las herramientas, así como al aprendizaje en el uso de muchas de ellas, en particular: Packer, Terraform, Ansible, Prometheus y Grafana.

Como se mencionó en la Sección 1.3, este TFG no parte completamente de cero, sino que en el momento de unirme al proyecto ya existía una infraestructura creada en AWS utilizando el paradigma IaC con la herramienta Terraform. La estructura inicial constaba de una región de AWS con cuatro VPCs, una para cada entorno: development (desarrollo), preproduction (preproducción), production (producción) y management (administración). Cada VPC cuenta con una subred pública donde levantar las instancias que se crearán a partir de una AMI personalizada, exceptuando la VPC de administración que también presenta una subred privada. Cabe mencionar que una subred pública es aquella en la que el tráfico se direcciona a un puerto de enlace a Internet (Internet gateway), mientras que una subred privada no tiene ninguna ruta a este puerto de enlace. Un gateway permite la comunicación entre una VPC e Internet, proporcionando un objetivo en las tablas de enrutamiento de la VPC para el tráfico

direccionable de Internet y realizando conversiones de las direcciones de red (NAT) para las instancias que tengan asignadas direcciones IPv4 públicas. Las instancias que se generan en cada entorno tienen definidos diferentes grupos de seguridad que han sido previamente creados. Un grupo de seguridad de AWS es un firewall a nivel de instancia que sirve para controlar su tráfico entrante y saliente.

Por un lado, las VPC de desarrollo, preproducción y producción tiene como funcionalidad levantar un determinado número de instancias en una subred pública con el agente de Prometheus instalado para habilitar así su monitorización. Mientras que, por otro lado, la VPC de administración tiene como objetivo levantar una instancia en una subred privada que tenga configurado un Prometheus server y Grafana para poder así monitorizar el resto de instancias de las otras VPCs. Como se comentó anteriormente, dentro de una VPC las instancias que se encuentran en una subred privada no pueden enviar tráfico directamente a Internet. Para ello se recurrirá a la utilización de un bastión. Un bastión es una instancia de servidor especial que esta diseñada para ser el punto de acceso principal desde Internet y actúa como proxy para sus otras instancias. Esta nueva instancia se levanta en una subred pública de la VPC de administración dando acceso a Internet a la subred privada durante el tiempo que esté en funcionamiento.

Todas las instancias se crean a partir de una AMI personalizada (basada en Ubuntu 14) generada con Packer, donde uno de los elementos que se configuran a través de esta herramienta es el agente de Prometheus que llevan preconfigurado todas las instancias para la futura obtención de series de datos temporales. Partiendo de esto, la instancia inicial de la subred privada de la VPC de administración no tiene instalado ni el Prometheus server ni Grafana, por lo que el bastión es el encargado de realizar su aprovisionamiento, puesto que se necesita conexión a Internet para ello. A grandes rasgos, el bastión, que también tiene Ansible instalado, ejecuta un playbook que descarga un contenedor Docker (previamente creado) que dispone de todo lo necesario para que el Prometheus server y Grafana funcionen en la instancia destino. En la Figura 4.1 se muestra el diagrama del diseño inicial de la infraestructura. En esta figura se puede observar el rango de direcciones IPv4 asociado a cada VPC definido como bloque de direccionamiento entre dominios sin clases (CIDR). Como se muestra en la Tabla 4.1, los primeros 16 bits se emplean para identificar la red y los 16 bits restantes determinan el número de hosts (/16).

VPC	CIDR
administración (mng)	10.0.0.0/16
desarrollo (dev)	10.1.0.0/16
preproducción (pre)	10.2.0.0/16
producción (pro)	10.3.0.0/16

Tabla 4.1: Asignación CIDR para cada VPC

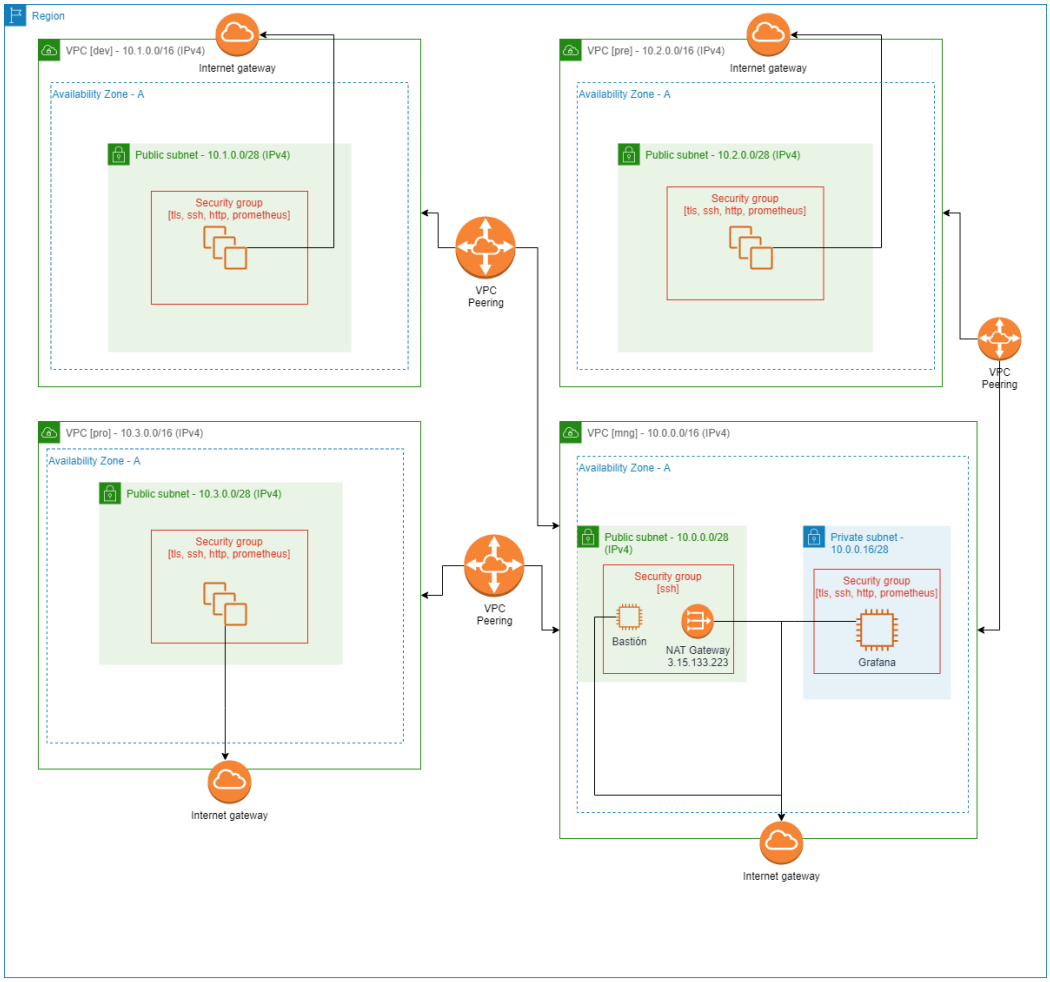


Figura 4.1: Diseño de la infraestructura inicial en AWS

A nivel de instancia, en la Figura 4.1 también se muestran los grupos de seguridad que contienen, permitiendo el tráfico a través de los puertos 80 (HTTP), 22 (SSH), 443 (TLS) o 9001 (Prometheus). El puerto 9001 se configura por defecto en todas las instancias para transmitir las métricas de Prometheus y en la instancia en la que está el servidor Prometheus se configura también como puerto receptor.

Para que exista comunicación entre las distintas VPCs hay que configurar una interconexión (peering) entre ellas. El VPC Peering es una conexión de redes entre dos VPCs localizadas en la misma red, que permite direccionar tráfico entre ellas utilizando la propia infraestructura existente de una VPC para crearla. Durante este proceso, la VPC de administración solicita la interconexión al resto (de una en una). Esto se ve reflejado en las flechas del VPC Peering en la Figura 4.1 donde la VPC de administración se conecta con las demás. Esta interconexión será utilizada para transmitir las métricas de Prometheus hasta la VPC de administración.

Por último, se puede observar en la Figura 4.1 que en la VPC de administración aparece representado el bastión en una nueva subred pública. La instancia bastión en sí no es capaz de proporcionar Internet a las subred privada, si no que lo consigue gracias al elemento NAT Gateway que se crea en el mismo archivo de configuración Terraform que levanta la instancia. El NAT Gateway principalmente permite a las instancias de la subred privada conectarse a Internet a la vez que impide a Internet iniciar una conexión con esas mismas instancias. Además, el bastión solo pertenece al grupo de seguridad SSH, puesto que su función principal es conectarse a las instancias de las posibles subredes privadas.

De este modo, la forma de acceder a Grafana sería a través de un túnel SSH, creado con MobaXterm, entre el bastión y la instancia que contiene Grafana de la subred privada por un puerto determinado. En la Figura 4.2 se muestra el túnel SSH creado con MobaXterm. Primero, se define el puerto a partir del cual se accede mediante el navegador web (9999 en este caso). Segundo, se indican los datos necesarios para que el bastión se conecte con la instancia que contiene Grafana, concretamente: la IP del bastión, el nombre de usuario de la instancia de Grafana y el puerto por el que se produce la conexión. Tercero y último, se indica la IP de la instancia Grafana y el puerto a través del cual funciona esta aplicación (5001 en este caso). Además, se debe configurar previamente la clave SSH para acceder a las instancias.

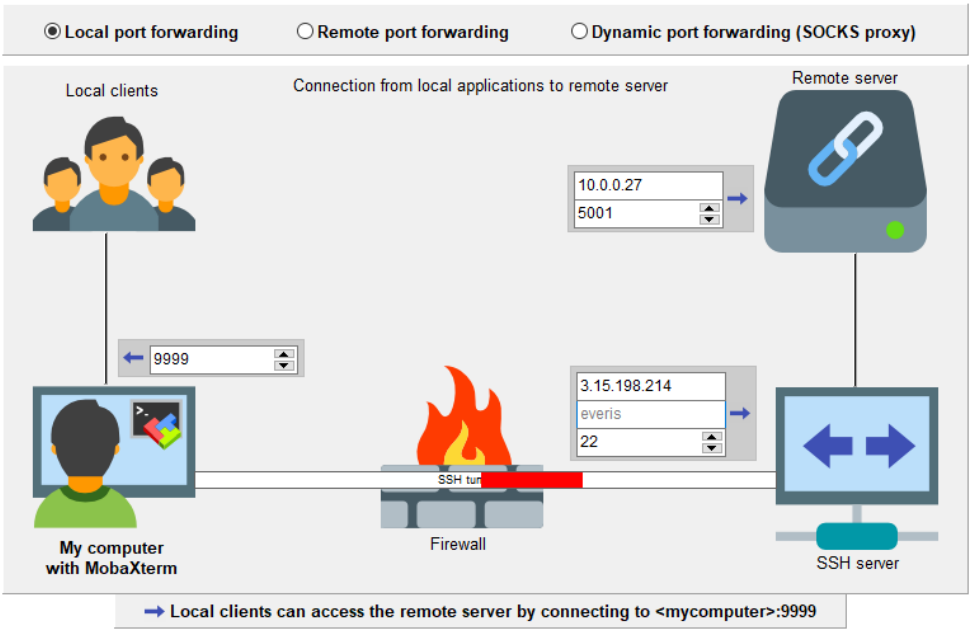


Figura 4.2: Túnel SSH para conectarse a Grafana

A lo largo de este capítulo se explica en más en detalle todos los componentes concretos que hacen posible la creación de esta infraestructura en AWS, así como las modificaciones realizadas a la misma.

4.2 Sprint 1 - Diseño e implementación de la infraestructura

En esta sección se explica el diseño e implementación de la nueva infraestructura en AWS a partir de las tareas obtenidas tras la reunión de planificación del Sprint 1, que se muestran en la Figura 4.3.

Sprint	Id tarea	Nombre de la tarea	Inicio	Fin	Esfuerzo	Coste (€)
1	1	Adaptación al estado actual del proyecto	05/10/2020	06/10/2020	4	92,52
1	2	Configuración de las instancias base de las máquinas (AMI personalizada) con Packer	07/10/2020	12/10/2020	6	158,78
1	3	Nuevo diseño de la infraestructura	13/10/2020	13/10/2020	2	46,26
1	4	Nueva estructuración del repositorio de Git	14/10/2020	14/10/2020	1	53,13
1	5	Refactorización y modificaciones del código de Terraform para implementar la nueva infraestructura a partir del nuevo repositorio de Git	14/10/2020	28/10/2020	16	490,08
1	6	Pruebas del funcionamiento de la infraestructura	28/10/2020	29/10/2020	3	79,39
				Total	32	920,16

Figura 4.3: Planificación y estimación del Sprint 1

4.2.1 Tarea 1 - Adaptación al proyecto

Durante el tiempo de desarrollo de esta tarea, se llevó a cabo el estudio del diseño de la infraestructura inicial y de los elementos que la forman, así como la configuración de los mismos a través de los archivos de Terraform. También se revisó la estructura existente en el repositorio de Git donde se almacenan todos los archivos que forman esta infraestructura.

4.2.2 Tarea 2 - AMI personalizada con Packer

El objetivo principal que se quiere conseguir con Packer es crear una imagen máquina (AMI) que contenga por defecto el agente Prometheus para producir datos de series de tiempo de las futuras instancias. En un primer instante, existía una AMI personalizada creada a partir de Ubuntu Server 14, pero se optó por eliminarla y definir una nueva desde cero debido a que el template se encontraba desactualizado y no era funcional. Aparte de implementar una funcionalidad para AWS, también se contempla la creación de una imagen máquina idéntica para VirtualBox con el fin de poder realizar pruebas en local antes de desplegarla en el cloud.

La nueva imagen máquina que se desea crear es una Ubuntu Server 18.04.2 con una CPU, 1024 GB de memoria RAM y 8 GB de espacio de almacenamiento. Los builders que se usan en el archivo de configuración de Packer son *amazon-ebs* para generar la AMI personalizada y *virtualbox-iso* para crear el OVF de la máquina virtual de VirtualBox. Por un lado, el builder *amazon-ebs* crea una AMI lanzando una instancia EC2 desde una AMI de base, aprovisionando dicha máquina en ejecución y luego creando una nueva AMI desde ella. Para este proceso el

propio builder se encarga de la creación de un par de claves temporales, reglas de grupos de seguridad, etc. que proporcionan acceso a la instancia mientras se crea la AMI. Existen muchas opciones de configuración a destacar como el tipo de virtualización, en este caso Hardware Assisted Virtualization (HVM) para proporcionar un hardware totalmente virtualizado, o *user data file* que se utiliza para configurar los datos de usuario. También es obligatorio indicar la región de AWS donde se desea crear la AMI, así como las credenciales de usuario. Por otro lado, el builder *virtualbox-iso* genera un archivo en formato OVF a partir de una imagen ISO. Para que la ejecución sea satisfactoria se le debe indicar como parámetro la secuencia de comandos de arranque.

Con todo lo anterior, se obtiene la AMI (y OVF) con una instalación básica de Ubuntu, por lo que a mayores se recurre a provisioners para instalar el agente Prometheus como nodo exportador, a través de un playbook de Ansible. Aunque Ansible se suele utilizar para aprovisionar un host a partir de otro, en este caso se usa para aprovisionar al mismo host gracias al provisioner *ansible-local*. Pero antes de ello es necesario instalar Ansible, por lo que previamente se ejecuta un script utilizando el provisioner *shell* que ejecuta todos los comandos de terminal necesarios para su instalación.

Seguidamente, se ejecuta el provisioner *ansible-local* al que se le pasan como parámetros el archivo del playbook a ejecutar (*main.yml*) y el archivo necesario para Ansible Galaxy (*requirements.yml*), ya que la instalación de Prometheus se corresponde con un rol disponible en el repositorio Galaxy. En el archivo *requirements.yml* se indican los cuatro roles del repositorio Galaxy que ejecuta el playbook para instalar: el nodo exportador de Prometheus, Java, un firewall para configurar reglas iptables en las instancias y un último rol que realiza configuraciones de seguridad para conexiones SSH. En el playbook *main.yml* se indica la ejecución de estos cuatro roles y se añaden los parámetros correspondiente a los mismos. Además, se indica el host al que va dirigida la ejecución del playbook (localhost en este caso) y se le proporciona privilegios de superusuario con el parámetro ”become”. Por último a través de otro script con el provisioner *shell* se desinstala Ansible y sus dependencias de la instancia. Una vez que esto finaliza, la instancia que se creó se destruye y ya tenemos nuestra nueva AMI lista en AWS o nuestro OVF en local para VirtualBox. En el Anexo A se añade el template final generado y algunos ficheros de configuración.

4.2.3 Tarea 3 - Nuevo diseño de la infraestructura

Al iniciar el TFG se consideró que la infraestructura era bastante funcional, pero que se podían mejorar determinados aspectos. En cuanto a las VPCs se decidió añadir una capa adicional de seguridad implementando Access Control List (ACL) a nivel de VPC. Una lista de control de acceso o ACL de red es una capa de seguridad que actúa como firewall de tráfico entrante y saliente de una o varias subredes. Su funcionamiento es similar al de los grupos

de seguridad pero trabajan en una capa superior. Para las VPCs de desarrollo, preproducción y producción se consideró que lo que había implementado hasta ahora era suficiente para cumplir los objetivos marcados en el proyecto. Sin embargo, para la VPC de administración se decidió implementar una forma más óptima que permita acceder a los dashboards de Grafana, que como se comentó en la Sección 4.1, actualmente se realizaba a través de un túnel SSH. En una primera aproximación surgió la idea de crear un servidor proxy en una instancia de una subred pública para implementar el acceso a Grafana (véase Figura 4.4).

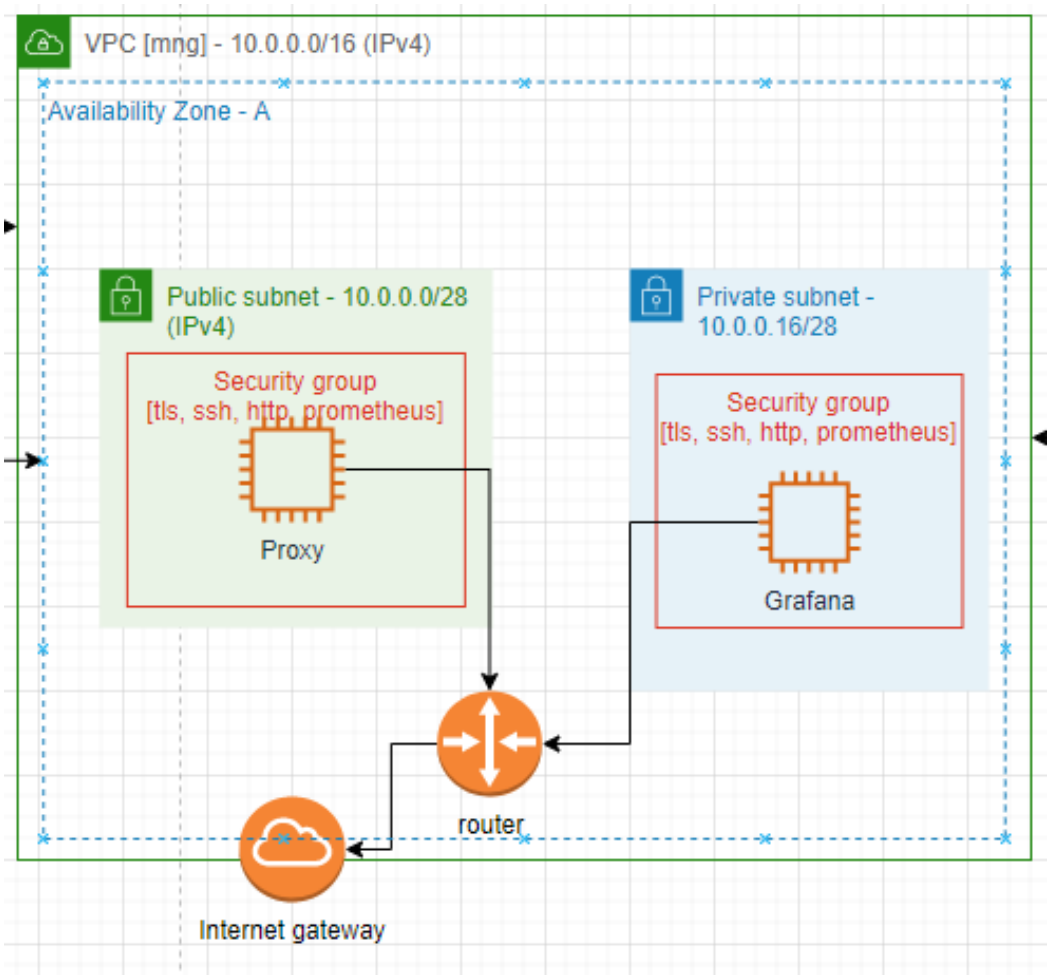


Figura 4.4: Diseño del servidor proxy

Sin embargo, finalmente se optó por comprar un domino de Internet y utilizar desde AWS un Elastic Load Balancing (ELB) que permite distribuir el tráfico entrante de un sitio web entre las instancias. De esta manera se configura un puerto para enrutar el tráfico que esté admitido en los grupos de seguridad de la instancia de Grafana y en el ELB el puerto para escuchar. Posteriormente, el servicio de AWS denominado Route 53 [34] se encarga de realizar un ser-

vicio web de DNS entre el ELB y el usuario final que accede a través del dominio al servicio de Grafana. En la Figura 4.5 podemos ver gráficamente como quedaría distribuida la VPC de administración con este nuevo elemento además de las ACL anteriormente mencionadas.

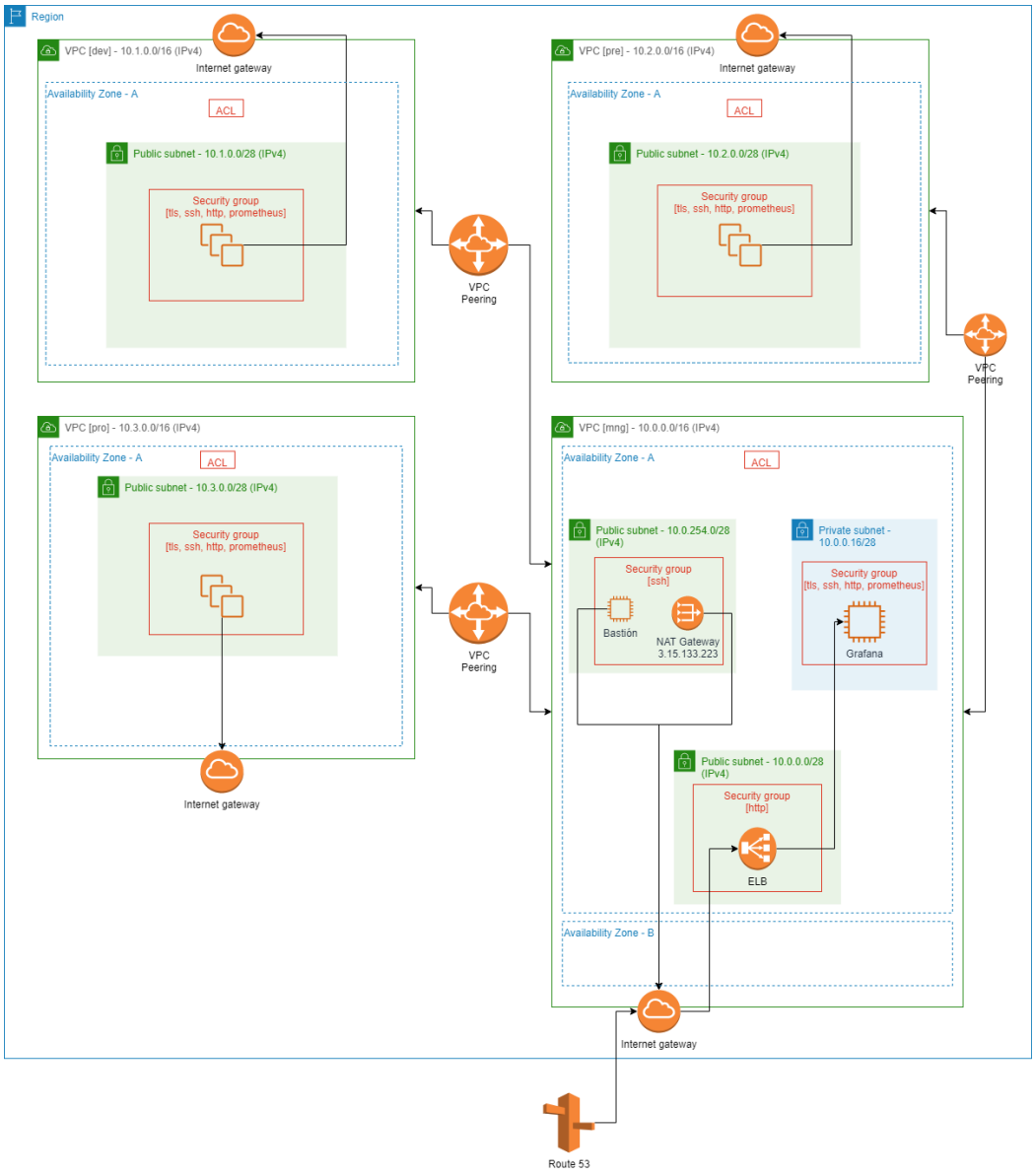


Figura 4.5: Diseño final de la infraestructura en AWS

Además, el aprovisionamiento de la instancia de la subred privada de la VPC de administración ya no lo realiza el bastión, si no que se crea una nueva instancia para ello. Esta se levanta, aprovisiona y se destruye, eliminando así código innecesario en el bastión.

4.2.4 Tarea 4 - Nueva estructuración del repositorio Git

Una vez decidido el nuevo diseño de la infraestructura, se ha de modificar el código siguiendo el paradigma IaC con la herramienta Terraform. Pero antes de ello, se determina que la estructuración actual del repositorio Git no es la más apropiada. En un primer momento existían, para el caso concreto de Terraform, hasta seis carpetas donde dentro de cada una se implementaba una funcionalidad de la infraestructura. Estas seis carpetas eran:

- terraform-module-aws-vpc: Repositorio con la información para crear las cuatro VPCs.
- terraform-module-aws-vpc-peering: Repositorio con la información del VPC Peering.
- terraform-network: Repositorio donde se creaba el bucket S3 y las VPC con sus conexiones (clonando los dos repositorios previos).
- terraform-infraestructure-mng: Repositorio que levanta la infraestructura propia de la VPC de administración.
- terraform-infraestructure-bastion: Estructura del bastión y sus elementos.
- terraform-module-aws-instance: Repositorio que contiene la información para crear instancias a partir de la AMI personalizada generada con Packer.

Como esto obligaba a realizar múltiples llamadas a Git dentro de la implementación, se optó por crear un nuevo repositorio denominado *terraform-production* que estuviera formado por la siguiente estructura de directorios, donde dentro de cada uno se encuentran uno o varios archivos de configuración HCL para la creación de los elementos con Terraform:

- /cross
  - /instance: Código necesario para crear una o varias instancia a partir de la AMI personalizada generada con Packer.
  - /peering: Código para establecer el VPC Peering.
  - /vpc: Código para crear una VPC.
- /epehemeral : Aparte de contener los siguientes directorios, tiene un archivo HCL que ejecuta /cross/vpc para levantar las VPCs.
  - /global: Código que ejecuta /cross/peering para crear el VPC peering.
  - /infrastructure-mng: Código de creación de la subred privada, la instancia que contendrá Grafana y los elementos necesarios para el ELB.
  - /provisioning-mng-instance: Instancia de aprovisionamiento de la instancia de la subred privada que contendrá Grafana.

- /keys: Código de creación del par de claves SSH para acceder a las instancias.
- /permanent: Código de creación del bucket S3 y su tabla DynamoDB asociada.

De este modo, en un único repositorio se encuentran todos los archivos necesarios para desplegar la infraestructura. El único repositorio que sigue manteniéndose separado es el bastión, puesto que es un elemento que se crea y se destruye según sea necesario. En la Figura 4.6 se muestra la estructuración final del repositorio Git, donde dentro de */projects* se encuentra el código del CLI, en *ansible-library* los archivos para aprovisionar la instancia de Grafana, en *terraform-library* la carpeta *terraform-production* y *terraform-bastion*, en *packer-library* los archivos para crear la AMI y en *docker-library* el contenedor Docker que se utiliza para instalar Prometheus y Grafana en la instancia correspondiente.

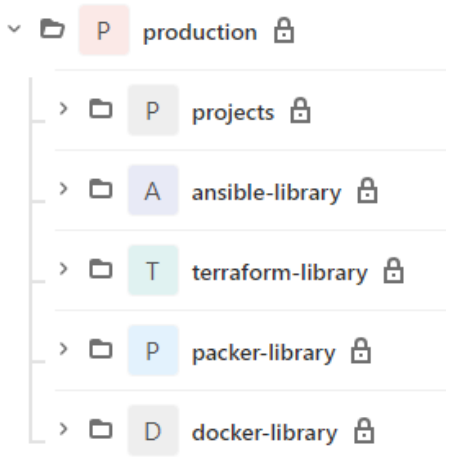


Figura 4.6: Estructura final del repositorio Git

4.2.5 Tarea 5 - Refactorización del código de la infraestructura

En esta tarea se explica la refactorización realizada del código Terraform para la nueva infraestructura siguiendo la estructura del repositorio Git *terraform-production*, explicado en la Sección previa (4.2.4). Se realiza una división en directorios para explicar la función de cada uno de ellos.

Directorio /permanent

Como se explicó en la Sección 2.2, en Terraform los archivos HCL se ejecutan (*terraform apply*) conjuntamente por directorios y previa creación de un fichero de estado (*terraform init*) que almacena la información de qué elementos se van a crear. Por ello, lo primero que hay que definir en la nueva estructura es el directorio ***/permanent*** donde se implementa los recursos

de Terraform: *aws_s3_bucket* y *aws_dynamodb_table*. El recurso *aws_s3_bucket* se encarga de crear el bucket S3 donde se almacenan los ficheros de estado de cada ejecución de Terraform y el recurso *aws_dynamodb_table* crea una base de datos DynamoDB asociada al bucket S3 para llevar el registro al bloquear un estado. También aparece el bloque de provider *aws* para indicar la API con la que trabaja Terraform, y dentro de él se indica la región de AWS donde se creará la infraestructura (us-east-2 en este caso). Este provider se repite en un archivo HCL de cada directorio para indicar esto mismo.

Antes de continuar con los siguientes directorios, existe otro elemento que se va a repetir en los archivos de configuración, menos en */permanent*, del mismo modo que el provider *aws*. Este elemento es el backend, que define cómo se realizan las operaciones y donde se almacena el estado. Concretamente, en esta infraestructura se utiliza el backend para indicar donde se almacena el estado en el bucket S3. En el fragmento de código 4.1 se puede observar este elemento junto a sus parámetros. Con el parámetro "bucket" se indica el nombre del bucket S3 donde se va a almacenar, "key" representa la ruta al fichero de estado dentro del bucket que cambia en función del directorio, con "region" la región de AWS donde se encuentra el bucket, "dynamodb_table" indica el nombre de la tabla DynamoDb asociada, y con "encrypt" se implementa encriptación para los ficheros de estado en el lado del servidor.

```
terraform {
  backend "s3" {
      # No variables allowed here
      bucket         = "terraform-produc-state"
      key            = "ruta_directorio/terraform.tfstate"
      region         = "us-east-2"
      dynamodb_table = "terraform-produc-locks"
      encrypt        = true
  }
}
```

Listing 4.1: Elemento Terraform backend S3

En Terraform también se pueden utilizar variables dentro de los ficheros de configuración, para ello existen dos maneras de hacerlo: de forma local en un archivo HCL mediante el bloque local, o creando un archivo variables.tf donde definirlas. Además, existe un bloque especial denominado data sources o fuentes de datos, que permite obtener o calcular datos para su uso en cualquier otro lugar de la configuración de Terraform.

Directorio /keys

En el directorio ***/keys*** se crean, a partir de un script, el par de claves pública/privada que se utilizan para conectarse mediante ssh a las instancias. Se registran en AWS mediante el recurso *aws_key_pair* indicándole el nombre de la clave para AWS y el fichero con el contenido de

la clave pública. A la hora de subir esta información a Git no se incluyen los ficheros con las claves por motivos de seguridad.

Directorio /cross

La carpeta ***/cross*** está formada por tres subcarpetas: ***/vpc***, ***/peering*** y ***/instance***. Ninguna de ellas contiene el elemento backend, debido a que están diseñadas para funcionar como módulos a los que llamar desde otro archivo HCL.

La subcarpeta ***/cross/vpc*** contiene todos los archivos de configuración necesarios para crear una VPC, consiguiendo así mayor escalabilidad de la que existía antes. En estos archivos se crea la VPC, con su Internet gateway, las reglas ACL, los grupos de seguridad que podrán usar sus instancias y la tabla de rutas. La VPC se genera a partir del recurso *aws_vpc*, donde debemos indicar el CIDR como parámetro obligatorio y, como parámetros opcionales, se ha decidido incorporar soporte DNS en la VPC para que las instancias con direcciones IP públicas obtengan los nombres DNS públicos correspondientes. En el fragmento de código 4.2 se muestra este último aspecto, donde el parámetro "tags" se usa para darle formato al nombre del elemento, así en el caso de que se acceda al recurso mediante la aplicación web de AWS se podrá identificar sin problemas. El valor de los parámetros se indica como variable para poder escoger los pertinentes en función de las características que le queramos dar a la VPC. Esto último se repetirá, sobre todo a lo largo de los archivos HCL de estos tres directorios por su finalidad de ser utilizados como módulos.

```
resource "aws_vpc" "main" {
  cidr_block = var.cidr
  enable_dns_hostnames = var.enable_dns_hostnames
  enable_dns_support = var.enable_dns_support

  tags = {
    Name = format ("%s-vpc", var.name)
  }
}
```

Listing 4.2: Recurso para crear una VPC

El Internet gateway se crea con el recurso *aws_internet_gateway*, indicándole como parámetro el identificador de la VPC. Para las reglas de ACL se utiliza *aws_default_network_acl* que permite seleccionar primero si es de entrada o de salida y después el número de regla, puerto de origen y destino, protocolo, CIDR y la acción (permitir/denegar). También se definen con variables para que luego cada VPC pueda añadir las reglas que desee. Los grupos de seguridad que pueden escoger las instancias se definen con el recurso *aws_security_group*. Se han creado grupos de seguridad para permitir: tráfico entrante desde cualquier IP (0.0.0.0/0) para TLS a través del puerto 443, tráfico entrante HTTP por el puerto 80, tráfico de Prometheus por el

puerto 9100, y tráfico SSH por el puerto 22. También se crean los mismos grupos de seguridad pero que sólo permiten tráfico entrante desde cualquier VPC (10.0.0.0/8) o solamente dentro de una VPC (10.0.0.0/16). Por último, para la VPC se crea su tabla de rutas con el recurso *aws_route_table*, indicando que el tráfico con destino 0.0.0.0/0 se dirija al Internet gateway, y con el recurso *aws_main_route_table_assocaition* la asociación de esa tabla con la VPC.

En el subdirectorio ***/cross/peering*** se encuentra el archivo de configuración HCL para crear la interconexión entre las VPCs y así poder comunicarse entre ellas. Primero, con el recurso *aws_vpc_peering_connection* se indican el identificador de la VPC que solicita crear la interconexión y el identificador de la VPC destino, y para administrar el lado que acepta la interconexión se recurre al recurso *aws_vpc_peering_connection_accepter*. Con el recurso *aws_vpc_peering_connection_options* se añaden opciones de configuración para la VPC que acepta la interconexión. Concretamente, con el parámetro "allow_remote_vpc_dns_resolution" se permite que la VPC resuelva los nombres DNS públicos en direcciones IP privadas cuando se le consulte desde instancias en una VPC del mismo nivel. Por último, se añaden las rutas en las tablas de enrutamiento con el recurso *aws_route*, una en la tabla de la VPC que solicita la conexión indicando el bloque CIDR de destino y el identificador de la interconexión, y otra en la tabla de la VPC que acepta la conexión con los mismos parámetros.

Finalmente, en la subcarpeta ***/cross/instances*** se encuentra el código necesario en un archivo HCL para crear una o varias instancias en una subred pública o privada a partir de la AMI personalizada creada con Packer. La AMI ya existe en AWS por lo que para seleccionarla no se necesita un bloque de recursos, sino un bloque de datos. Con el data source *aws_ami* se obtiene el identificador de la AMI que necesita el recurso que crea la instancia (el fragmento de código 4.3 muestra este bloque). Se observa el parámetro "owners" que indica el identificador de la cuenta de AWS propietaria de la AMI y los filtros utilizados para encontrarla.

```
data "aws_ami" "selected" {
    most_recent = true
    filter {
        name = "name"
        values = [format("%s-%s-%s", var.os_type, var.os_version,
    var.os_arch)]
    }
    filter {
        name   = "virtualization-type"
        values = ["hvm"]
    }
    owners = [secuencia_de_dígitos]
}
```

Listing 4.3: Data source para obtener el AMI personalizada

Con el recurso *asw_network_interface* se crea el interfaz de red para las instancias. Destaca

el parámetro "security_groups" que recibe, a partir de una variable, el nombre de los grupos de seguridad que contendrán las instancias creadas. Estos grupos se obtienen de los creados en */cross/vpc/* con el recurso *aws_security_group*. Por último, con el recurso *aws_instance* se crea la instancia EC2 pasándole como parámetro la AMI personalizada, el nombre de la clave SSH, el estado en el que debe encontrarse cuando se termine (terminate), el número de instancias a crear, la asociación con el interfaz de red previamente creado y la configuración de los volúmenes de almacenamiento del dispositivo. También se añaden los recursos de datos *aws_vpc* para obtener el nombre de la VPC en la que se está creando, y *aws_subnet* para indicar en que subred se van a crear las instancias, por la finalidad de utilizar este archivo de configuración como módulo.

Directorio /ephemeral

En la carpeta ***/ephemeral*** se encuentra el archivo de configuración encargado de crear las VPCs a partir del código de */cross/vpc* y las subcarpetas ***/global***, ***/infrastructure-mng*** y ***/provisioning-mng-instance***. El archivo de creación de las VPCs simplemente llama, con el bloque module, al código anteriormente explicado de */cross/vpc*. En él se le indica el código base, el nombre que adopta la VPC, el bloque CIDR y la existencia opcional de alguna regla ACL. El fragmento de código 4.4 muestra como generar la VPC de desarrollo (dev). En este

```
module "vpc_dev" {
  source = "../cross/vpc"
  name = "dev"
  cidr = "10.1.0.0/16"

  inbound_rules = [
    {
      "rule_no"     = 100,
      "from_port"   = 0
      "to_port"     = 0
      "protocol"    = "-1"
      "cidr_block"  = "0.0.0.0/0"
    }
  ]
  outbound_rules = [
    {
      "rule_no"     = 100,
      "from_port"   = 0
      "to_port"     = 0
      "protocol"    = "-1"
      "cidr_block"  = "0.0.0.0/0"
    }
  ]
}
```

Listing 4.4: Módulo para la creación de la VPC dev

caso, el valor de "name=dev" y de "cidr=10.1.0.0/16" son los que se substituyen en "var.name" y "var.cidr" del archivo indicado con el parámetro "source=../cross/vpc", que se representó con anterioridad en el fragmento de código 4.2. Para el resto de las VPCs el código sería idéntico pero modificando los valores de "name" y "cidr". Además, se añade de nuevo el elemento backend S3 (véase Sección 4.1), modificando la ruta (key) donde se almacena el estado de Terraform.

Dentro de la subcarpeta ***ephemeral/global*** está el HCL que ejecuta el código de */cross/-peering* para generar la interconexión de VPCs. Como se quiere crear tres interconexiones bidireccionales (VPC mng con dev, mng con pre y mng con pro), se implementan tres bloques de módulo, uno para cada interconexión. En el fragmento de código 4.5 se muestra el caso concreto para establecer el VPC Peering entre las VPCs de mng y dev, donde se les asigna un valor a los parámetros que se comentaron al explicar la subcarpeta */cross/peering*. El valor de los parámetros "main_vpc_id" y "main_route_table_id" se obtiene a partir de un data source de *aws_vpc* y *aws_route_table*, respectivamente, que se almacena como variable local para luego pasársela al módulo. También se añade de nuevo el elemento backed S3, modificando la ruta donde se almacena el estado de Terraform.

```
module "dev-peering" {
  source = "../../cross/peering"
  name                 = "mng"
  enable_vpc_peering   = true
  main_vpc_id          = local.vpc_id
  main_region          = var.region
  peer_vpc_name        = "dev-vpc"
  peer_region          = var.region
  auto_accept_peering  = true
  main_dns_resolution  = true
  peer_dns_resolution  = true
  main_route_table_id  = local.main_route_table_id
  peer_route_table_name = "dev-route-table"
}
```

Listing 4.5: Módulo para el VPC Peering entre la VPC mng y dev

En el subdirectorio ***ephemeral/infrastructure-mng*** se encuentran los archivos de configuración para crear la infraestructura propia de la VPC de administración. El objetivo es crear una subred privada con la instancia vacía que contendrá Prometheus y Grafana. Para lograrlo, lo primero que se debe hacer es definir la subred privada en la que se levanta la instancia. Esto se consigue con el bloque de recurso *aws_subnet* indicando la VPC donde se levanta, el bloque CIDR de la subred, la zona de disponibilidad y si se desea asignar una dirección IP pública a las instancias que se levanten en la subred. Este último parámetro ("map_public_ip_on_launch") es el que determina si una subred es pública o privada. En el fragmento de código 4.6 se muestra el caso concreto de la subred creada para esta instancia. La obtención del "vpc_id" se hace

de la misma manera que para el módulo del VPC Peering.

```
resource "aws_subnet" "grafana" {
  vpc_id                  = local.vpc_id
  cidr_block              = "10.0.0.16/28"
  map_public_ip_on_launch = false
  availability_zone       = var.private_availability_zone
  tags = {
    Name = format(
      "%s-%s-private-subnet-grafana",
      local.env,
      var.private_availability_zone
    )
  }
```

Listing 4.6: Recurso para crear la subred de Grafana

Lo segundo es definir el módulo que llama al código de */cross/instance* que genera la instancia. En el fragmento de código 4.7 se muestra este módulo junto a los parámetros que definen la instancia. Destacan el "subnet_id" para indicar la subred en la que se levanta, "instance_count" para indicar el número de instancias (una en este caso), y "security_groups" para determinar los grupos de seguridad.

```
module "aws_monitoring_instance" {
  source = "../../cross/instance"
  enable = true
  subnet_id = aws_subnet.grafana.id
  instance_count = 1
  region = var.aws_region
  os_type = var.os_type
  os_version = var.os_version
  os_arch = var.os_arch
  env = local.env
  project = "monitoring"
  instance_type = "t2.micro"
  volume_size = 8
  private = true
  security_groups = [data.aws_security_group.vpc_tls.id,
    data.aws_security_group.vpc_ssh.id,
    data.aws_security_group.vpc_http.id,
    data.aws_security_group.vpc_prometheus.id]
}
```

Listing 4.7: Módulo para la instancia de Grafana

Como se puede observar, la etiqueta "vpc" señala que los grupos de seguridad escogidos restringen el tráfico a nivel de VPC. Las funcionalidades de estos grupos de seguridad son los que se muestran en la Tabla 4.2.

Lo tercero es configurar el dominio de la compañía para acceder a Grafana, y así sustituir

Reglas de entrada			
Nombre	**Tipo de protocolo**	**Número de puerto**	**IP origen**
vpc_tls	TCP	443	10.0.0.0/16
vpc_ssh	TCP	22	10.0.0.0/16
vpc_http	TCP	80	10.0.0.0/16
vpc_prometheus	TCP	9100	10.0.0.0/16
Reglas de salida			
Nombre	**Tipo de protocolo**	**Número de puerto**	**IP destino**
vpc_tls	TCP	443	0.0.0.0/0
vpc_ssh	TCP	22	10.0.0.0/16
vpc_http	TCP	80	0.0.0.0/0
vpc_prometheus	TCP	9100	10.0.0.0/16

Tabla 4.2: Grupos de seguridad de la instancia Grafana

definitivamente el túnel SSH a través del cual se accedía hasta ahora. Para ello, se configura el balanceador de carga con los recursos *aws_lb*, *aws_lb_target_group*, *aws_lb_listener* y *aws_lb_target_group_attachment*. Con *aws_lb* se define la entidad ELB indicándole su nombre, si es de uso interno o de cara a Internet, el tipo, las subredes donde se puede levantar y sus grupos de seguridad. En el fragmento de código 4.8 se muestra este recurso, donde se observa que es un ELB de tipo aplicación, el más adecuado para el tráfico HTTP, que puede ser levantado en dos subredes de diferentes zonas de disponibilidad (requisito del ELB), y que pertenece al grupo de seguridad que admite todo el tráfico (0.0.0.0/0) HTTP entrante y saliente por el puerto 80. Este grupo de seguridad es necesario ya que Grafana está configurado para que transmita a través del puerto 80 y el acceso al dominio se realizará por dicho puerto también.

```
resource "aws_lb" "lb" {
    name = "lb-tf-grafana"
    internal = false
    load_balancer_type = "application"
    subnets = [aws_subnet.public_zA.id, aws_subnet.public_zB.id]
    security_groups = [data.aws_security_group.public_http.id]
}
```

Listing 4.8: Recurso para crear el ELB

Seguidamente, con el recurso *aws_lb_target_group* y *aws_lb_target_group_attachment* se define el destino al cual se direccionan las solicitudes utilizando el protocolo y el número de puerto especificados. En este caso, se direcciona a la instancia de Grafana y la configuración de *aws_lb_target_group* se puede observar en el fragmento de código 4.9, donde se determina el

puerto, el protocolo y la VPC en la que se encuentra. El recurso *aws_lb_target_group_attachment* se utiliza para asociar la instancia concreta de Grafana con lo proporcionado en el recurso *aws_lb_target_group*.

```
resource "aws_lb_target_group" "grafana" {
    name = "tf-grafana-lb-tg"
    port = 80
    protocol = "HTTP"
    target_type = "instance"
    vpc_id = data.aws_vpc.selected.id
}
```

Listing 4.9: Recurso para definir el objetivo del ELB

Posteriormente, con el recurso *aws_lb_listener* se comprueban las solicitudes de conexión mediante el protocolo y el puerto configurados, y se reenvían hasta su destino (target group). En el fragmento de código 4.10 se proporciona la definición de este recurso. En la Figura 4.7 se muestra un gráfico del funcionamiento de un ELB con sus recursos.

```
resource "aws_lb_listener" "grafana" {
    load_balancer_arn = aws_lb.lb.arn
    port = 80
    protocol = "HTTP"

    default_action{
        type = "forward"
        target_group_arn =aws_lb_target_group.grafana.arn
    }
```

Listing 4.10: Recurso del intermediario entre el objetivo y el ELB

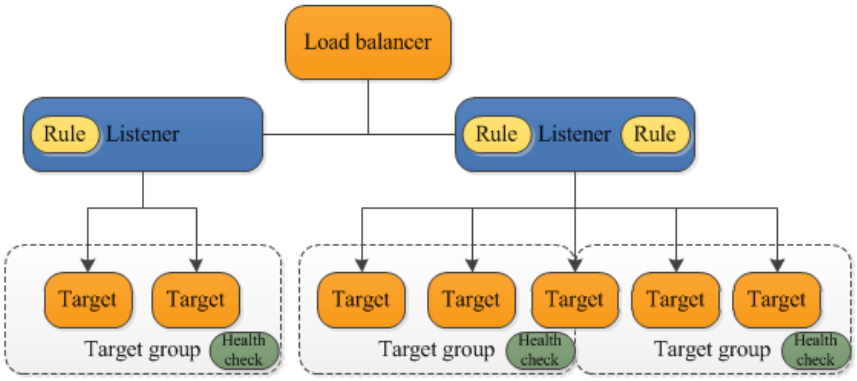


Figura 4.7: Estructura del ELB

Lo cuarto y último para el subdirectorio ***ephemeral/infrastructure-mng***, es utilizar el

recurso *aws_route53_record* para registrar el nombre de dominio y utilizarse como servidor DNS. En el fragmento de código 4.11 se muestra este recurso, donde se indica el identificador de la región (us-east-2) donde trabaja esta estructura, el nombre del dominio de la compañía, con "type=A" se indica que el DNS va a realizar conversión URL a IPv4, y en el "alias" se apunta la URL del dominio con el ELB y se comprueba si el estado del recurso es correcto antes de realizar la conversión.

```
resource "aws_route53_record" "name" {
    zone_id = var.zone_id_route53
    name = var.domain_name
    type = "A"

    alias{
        name = aws_lb.lb.dns_name
        zone_id = aws_lb.lb.zone_id
        evaluate_target_health = true
    }
}
```

Listing 4.11: Recurso para crear el Route53

En la subcarpeta ***ephemeral/provisioning-mng-instance*** se define el archivo de configuración HCL que genera la instancia que aprovisiona la instancia vacía de la subred privada de la VPC de administración. Para explicar este apartado se hace referencia como host a la instancia que aprovisiona, e instancia Grafana o destino a la instancia que se va a aprovisionar. Dentro de este archivo se crea una subred pública con el recurso *aws_subnet* con bloque CIDR 10.0.254.0/28. Dentro de esta subred se crea una instancia con el bloque módulo a partir del código de */cross/instance* indicando como grupo de seguridad "public_ssh", permitiendo el tráfico SSH por el puerto 22 de cualquier IP. De esta manera se puede acceder a esta instancia desde nuestra máquina vía SSH para enviarle diferentes archivos necesarios para el aprovisionamiento. También se crea un NAT Gateway, con el recurso *aws_nat_gateway*, para dar acceso a Internet a la instancia de la subred privada, ya que necesita este acceso para instalar algunos componentes. Antes de implementar este recurso se necesita definir una dirección IP elástica, que es una dirección IPv4 estática pública, con el recurso *aws_eip*. Al NAT Gateway se le pasa como parámetro el identificador de la dirección IP elástica previamente creada y el identificador de la subred donde se levanta.

Posteriormente, para que el NAT Gateway cumpla su función se actualiza la tabla de rutas de la subred privada para que apunte todo el tráfico vinculado a Internet a la NAT Gateway, como se muestra en la Figura 4.8. De este modo, la instancia de la subred privada envía el tráfico de Internet a la NAT Gatewayy este usa su dirección IP elástica como dirección IP de origen para enviar el tráfico al puerto de enlace de Internet (Internet Gateway). Con el recurso *aws_route_table*, se proporcionan los detalles de la ruta y con *aws_route_table_association* se

asocia a la tabla de enrutamiento de la subred privada. El NAT Gateway tiene acceso a Internet puesto que la tabla de enrutamiento asociada a la subred donde reside incluye una ruta que apunta el tráfico de Internet a un Internet Gateway, como se observa en la Figura 4.9.

Destination	Target
10.0.0.0/16	local
0.0.0.0/0	nat-02d5cf38f8d519f79

Figura 4.8: Tabla de rutas de la subred privada con el NAT Gateway

Destination	Target
10.0.0.0/16	local
10.1.0.0/16	pcx-0c120889d13a24848
10.2.0.0/16	pcx-0c04f48e7091c1b54
10.3.0.0/16	pcx-06721fcdc32086d7e
0.0.0.0/0	igw-092f037769a9bb178

Figura 4.9: Tabla de rutas de la subred pública donde se encuentra el NAT Gateway

En el diseño inicial, como se ya comento en la Sección 4.1, se obtenía la dirección IP del bastión para, a través de mobaXterm, conectarse vía SSH a él. Una vez dentro, usando SCP se copiaba el directorio que contiene los archivos de Ansible, se exportaban la clave privada SSH y como variables de entorno la claves de AWS, para luego ejecutar el playbook que instala Prometheus y Grafana a partir de una imagen de Docker. En esta nueva implementación para el aprovisionamiento, se crea un recurso *null_resource* que implementa el ciclo de vida de un recurso estándar pero no realiza ninguna otra acción. Este recurso "especial" sirve para ejecutar aprovisionadores que no están asociados directamente con un recurso específico, como es el caso. El código de este recurso especial se muestra en el fragmento de código 4.12.

Como se puede ver en el código, primero se establece la etiqueta "connection" que determina el host que se va a conectar vía SSH (y la clave) a la instancia que se quiere aprovisionar. Se observa como el host es la instancia que acabamos de crear, el nombre de usuario que se definió al crear la AMI con Packer, y la clave privada que se generó en el directorio */keys*. Después, aparece la etiqueta del provisioner file y remote-exec. La funcionalidad del primero es copiar el archivo indicado en el source (origen), que se encuentra en nuestra máquina, a la ruta indicada en el destination (destino), que se encuentra en la instancia host. Concretamente, copia la clave privada SSH a la máquina host y el script con la lista de instrucciones necesarias para el aprovisionamiento. Por otro lado, el provisioner remote-exec permite ejecutar comandos (o un script) en la instancia host. El aprovisionamiento se realiza a través de

```
resource "null_resource" "grafana_provisioner" {
  connection {
    type = "ssh"
    user = "everis"
    private_key = file("../../keys/key.pem")
    host = module.aws_ssh_instance.instance_ip[0]
  }
  provisioner "file" {
    source = "../../keys/key.pem"
    destination = ".ssh/key.pem"
  }

  provisioner "file" {
    source = "script.sh"
    destination = "/tmp/script.sh"
  }

  provisioner "remote-exec" {
    inline= [
      "git clone https://token@urlgit/production
                    /ansible-library/ansible-mng.git",
      "chmod +x /tmp/script.sh",
      "/tmp/script.sh",
      ]
  }
}
```

Listing 4.12: Recurso null_resource

Ansible, ejecutando un playbook desde el host con destino la instancia de Grafana. Es por ello que en remote-exec se clona el repositorio que contiene los archivos de Ansible para el aprovisionamiento. Posteriormente, se ejecuta el script.sh que contiene la lista de instrucciones que necesita la instancia host para instalar Prometheus y Grafana en la instancia destino a partir del playbook. Este script se muestra en el fragmento de código 4.13, donde además de algunas configuraciones previas necesarias, se observa la instalación de las dependencias de Docker en la máquina Grafana usando un rol de Asnbiel Galaxy y la ejecución del playbook con ansible-playbook. Como parámetro, se le indica al playbook el token que necesita para trabajar con el repositorio Git que contiene la imagen Docker para instalar Prometheus y Grafana. La imagen Docker y los archivos de Ansible utilizados se explican en la Sección 4.3, puesto que en el primer Sprint se decidió comprobar el funcionamiento de la monitorización tal y como estaba en la implementación inicial. Finalizado el proceso de aprovisionamiento, se destruye (*terraform destroy*) la instancia de aprovisionamiento junto a sus elementos. Con esto ya estaría levantada toda la infraestructura, pudiéndose acceder a los dashboards de Grafana mediante el dominio Web.

Por último, se mantiene separado en otro repositorio la instancia del bastión, que está compuesta de los mismos elementos que la instancia de aprovisionamiento exceptuando el

```
#!/bin/bash
chmod 400 .ssh/key.pem
cp .ssh/key.pem .ssh/everis
pip install boto3
export AWS_ACCESS_KEY_ID=
export AWS_SECRET_ACCESS_KEY=
cd ansible-mng/
ansible-galaxy install -r requirements.yml
ansible-playbook main.yml -e
    'ansible_python_interpreter=/usr/bin/python3' --extra-vars
    "git_token=token"
```

Listing 4.13: Script de aprovisionamiento para ejecutar Ansible

elemento *null_resource* y sus componentes. Simplemente, modificando el nombre de la VPC donde se quiera levantar se tendrá el bastión listo para poder conectarse vía SSH y dar acceso a Internet a las instancias de las posibles subredes privadas.

4.2.6 Tarea 6 - Pruebas

Una vez levantada toda la infraestructura se realizan pruebas funcionales para comprobar el correcto funcionamiento. Se accede mediante la aplicación web a la consola de AWS y se comprueba que todos los elementos están en perfecto estado. Posteriormente se accede al servidor de Grafana a través del dominio web y se levantan instancias vacías en las VPCs para asegurar que se envían métricas y son recibidas.

4.3 Sprint 2 - Diseño e implementación del sistema de monitorización

En esta sección se explica el diseño y la implementación del sistema de monitorización de la infraestructura a partir de las tareas obtenidas tras la reunión de planificación del Sprint 2, que se muestra en la Figura 4.10.

Sprint	Id tarea	Nombre de la tarea	Inicio	Fin	Esfuerzo	Coste (€)
2	1	Refactorización y modificaciones en el código proporcionado de Ansible para aprovisionar la instancia con Prometheus y Grafana	02/11/2020	03/11/2020	4	72,52
2	2	Actualización del Docker , Prometheus y Grafana	04/10/2020	09/10/2020	6	108,78
				Total	10	181,3

Figura 4.10: Planificación y estimación del Sprint 2

4.3.1 Tarea 1 - Refactorización del código de aprovisionamiento de Ansible

En esta sección se implementa el código de Ansible utilizado para el aprovisionamiento de la instancia de Grafana anteriormente explicado durante el Sprint 1. Se ha llevado a cabo la nueva codificación de los archivos de Ansible a partir de la existente eliminando dependencias innecesarias y añadiendo nuevas funcionalidades.

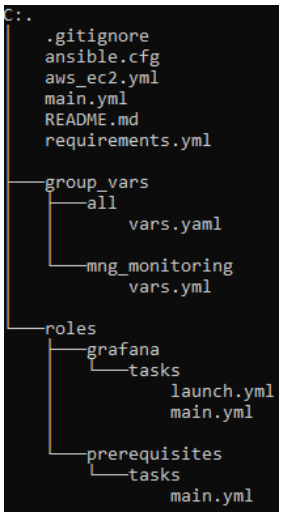


Figura 4.11: Estructura del repositorio de Ansible

En la Figura 4.11 se muestra la estructura del directorio y los ficheros de Ansible. Antes de ejecutar el playbook (*main.yml*), hay que configurar Ansible mediante *ansible.cfg* para indicarle el archivo de inventario (*aws_ec2.yml*), desactivar la verificación de la clave del host y habilitar los complementos que necesita. En el archivo *aws_ec2.yml* se indica que la instancia con la que se va a conectar pertenece a AWS y se le proporcionan los parámetros necesarios para la ejecución, como la región donde se encuentra y su nombre. También se indican el usuario y la clave SSH necesarios para la conexión.

El contenido del playbook *main.yml* se muestra en el fragmento de código 4.14, donde se indica el host, configurado en *aws_ec2.yml*, el usuario SSH para la conexión a partir de una variable configurada en */group_vars* y los tres roles a ejecutar. El rol "geerlingguy.docker", disponible en Ansbile Galaxy, realiza la instalación de Docker. El rol "prerequisites" ejecuta las tareas de que se encargan de instalar pip3 y con él instalar Docker Compose. Por último, el rol "grafana" (véase fragmento de código 4.15) contiene las tareas para clonar el repositorio Git con la imagen de Prometheus y Grafana, arrancar el demonio de Docker y ejecutar el *deploy compose* de la imagen. También se encarga de eliminar los archivos temporales de la instalación de Grafana.

```
- name: Launching monitoring
  hosts: mng_monitoring
  user: "{{ ssh_user }}"
  become: yes
  roles:
    - geerlingguy.docker
    - prerequisites
    - grafana
```

Listing 4.14: Playbook main.yml para instalar Prometheus y Grafana

```
- name: Clone git repo
  git:
    repo: https://{{git_token}}@git-url/production
    /docker-library/docker-compose-grafana.git
    version: master
    dest: "/tmp/docker-compose-grafana"
  become: yes

- name: Start docker daemon
  service:
    name: docker
    state: started
  become: true

- name: Deploy compose
  docker_compose:
    project_src: "/tmp/docker-compose-grafana"
    state: present

- name: Remove temp files
  file:
    path: "/tmp/docker-compose-grafana"
    state: absent
  become: yes
```

Listing 4.15: Tareas del rol Grafana

4.3.2 Tarea 2 - Docker, Prometheus y Grafana

Prometheus y Grafana se instalan y ejecutan a través de un contenedor Docker. Este contenedor está compuesto por dos archivos Dockerfiles uno para Prometheus y otro para Grafana, que se ejecutarán conjuntamente gracias a la herramienta Docker Compose. Por un lado, tenemos el Dockerfile correspondiente a Prometheus que se muestra en el fragmento de código 4.16. Con la instrucción “FROM” se establece la imagen base a utilizar y con ”COPY” se agrega el archivo *./conf/prometheus.yml* del directorio actual al cliente Docker.

En el archivo *prometheus.yml* se proporciona la configuración necesaria para que el servidor Prometheus, que está dentro de la instancia Grafana, reciba las series de datos de tiempo de las otras instancias de las diferentes VPCs. El archivo está escrito en formato YAML y se

```
FROM prom/prometheus:v2.12.0
COPY ./conf/prometheus.yml /etc/prometheus/prometheus.yml
```

Listing 4.16: Dockerfile de Prometheus

divide en contextos, que se muestran en el fragmento de código 4.17. Se puede observar que se exponen el contexto "global" y el "scrape_config" que configura la VPC dev (el resto de VPCs son iguales modificando el nombre). El primero que se utiliza es "global" que determina unos parámetros de configuración que son comunes para todos los contextos. Dentro de él se indica cada cuanto tiempo se extraen métricas. Seguidamente, con el contexto "scrape_configs" se definen la lista configuraciones para la extracción de series de datos temporales de los objetivos. En este contexto, la etiqueta "job_name" permite asignar un nombre por defecto al job, como el objetivo que envía las métricas se encuentra en el entorno de AWS, se necesita utilizar la etiqueta "ec2_sd_configs" para indicarlo. Con esta etiqueta se proporciona el listado de configuraciones de descubrimiento de servicios EC2, que permite recuperar series de datos temporales de instancias EC2. La dirección IP privada se usa de forma predeterminada para descubrir la instancia, pero se puede modificar por otros parámetros con el reetiquetado (relabeling). En primera instancia, como parámetros de "ec2_sd_configs" se indica en que región están los objetivos, las claves AWS para acceder al entorno y el puerto en el que se extraen las métricas (9100). Como no se desea extraer series de datos temporales de una instancia concreta, si no de todas las que se encuentran dentro de una VPC, se procede a realizar un reetiquetado ("relabel_configs") para configurarlo. De esta manera el servidor Prometheus espera recibir series de datos temporales a través del puerto 9100 de una instancia de la VPC.

```
global:
  scrape_interval: 1s
scrape_configs:
    - job_name: 'dev'
    ec2_sd_configs:
      - region: us-east-2
        access_key:
        secret_key:
        port: 9100
    relabel_configs:
    - source_labels: [__meta_ec2_tag_Environment]
      regex: dev.*
      action: keep
    - source_labels: [__meta_ec2_tag_Name]
      action: replace
      target_label: instance
```

Listing 4.17: Archivo de configuración Prometheus.yml

Por el otro lado, tenemos el Dockerfile de Grafana que es muy similar al de Prometheus,

mostrado en el fragmento de código 4.18. Dentro del directorio *./provisioning* existen dos carpetas: */datasource*, para configurar la fuente de datos, y */dashboard*, para crear el cuadro de mandos o dashboard. Esto se podría realizar desde la aplicación web de Grafana pero aquí se muestra el proceso de su automatización. Primero, en */datasource* se configura todo lo necesario para que Grafana conozca cual es la fuente de datos. El fragmento de código 4.19 muestra el archivo, donde se le indica el nombre que tendrá el data source en los paneles de Grafana, el origen o tipo de la fuente de datos, y la URL del servidor Prometheus, entre otras cosas.

```
1 FROM grafana/grafana:7.3.1
2 COPY ./provisioning /etc/grafana/provisionin
```

Listing 4.18: Dockerfile de Grafana

```
1 apiVersion: 1
2
3 datasources:
4 - name: EC2 - Prometheus
5   type: prometheus
6   url: http://grafana-prometheus-server:9090
7   editable: true
8   access: proxy
```

Listing 4.19: Datasource de Grafana

Segundo, en */dashboard* se genera el dashboard con los datos a monitorizar. Grafana cuenta con un gran repositorio de dashboards pero para este TFG se ha decidido realizar uno desde cero. Dentro de este directorio nos encontramos con los archivos *dashboard.yml* y *prometheus-node-exporter.json*. A través de dashboard.yml se configura un "provider" que se encarga de proporcionar a Grafana la información necesaria para crear un dashboard. El fragmento de código 4.20 muestra que se debe indicar la ruta al fichero donde está la configuración del dashboard y el formato del mismo. Dentro del archivo *prometheus-node-exporter.json* se encuentra toda la configuración necesaria para crear el dashboard que monitoriza el nivel de carga de CPU, de memoria RAM, la utilización del disco (tiempo de actividad) y el uso del sistema de ficheros y tiempo previsto para su llenado. En la Figura 4.12 se muestra el dashboard de Grafana con la información referente a una instancia de un entorno.

Por último, para definir y arrancar las dos imágenes al mismo tiempo se recurre a Docker Compose. En el archivo *docker-compose.yml* con la etiqueta "services" se indican los tres servicios que se van a iniciar, que son: Grafana, Prometheus y Database. El fragmento de código 4.21 muestra el contenido del archivo *docker-compose.yml* para los servicios de Grafana y Prometheus. Se puede ver que el puertos expuesto es el 80 y que se mapea al 3000 del contenedor, por ello en el balanceador de carga (ELB) se configuró el recurso *aws_lb_target_group* en el

```
apiVersion: 1

providers:
  # <string> an unique provider name
- name: 'JSON'
  # <int> org id. will default to orgId 1 if not specified
  orgId: 1
  # <string, required> name of the dashboard folder. Required
  folder: ''
  # <string, required> provider type. Required
  type: file
  # <bool> disable dashboard deletion
  disableDeletion: true
  # <bool> enable dashboard editing
  editable: true
  # <int> how often Grafana will scan for changed dashboards
  updateIntervalSeconds: 10
  options:
    # <string, required> path to dashboard files on disk. Required
    path: /etc/grafana/provisioning/dashboards/jso
```

Listing 4.20: Dashboard Provider de Grafana

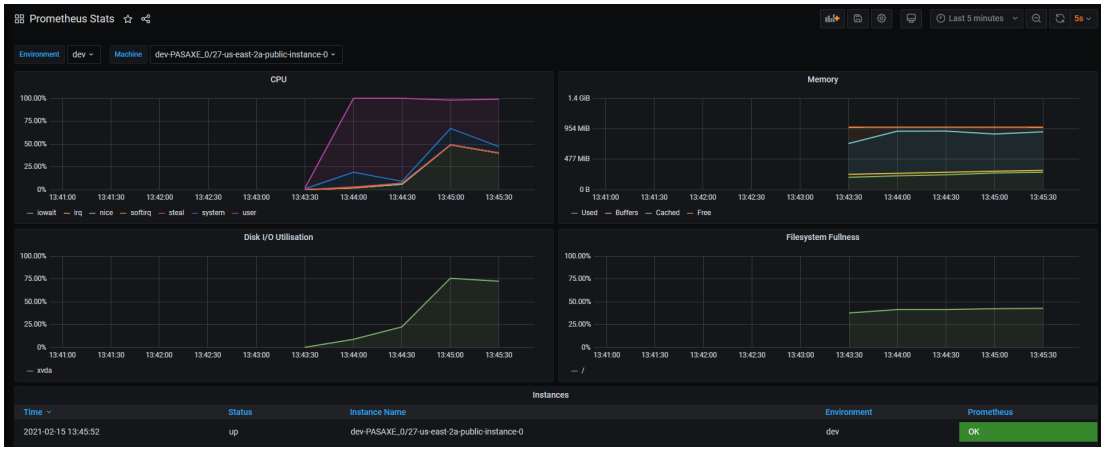


Figura 4.12: Dashboard de Grafana

puerto 80 (véase fragmento de código 4.9 y Sección 4.2.3).

En cuanto el servicio Database, se decide crear para tener una base de datos MySQL asociada a Grafana, obteniendo la imagen correspondiente de DockerHub para su creación. En el fragmento de código 4.22 se observa la parte del código del servicio Database perteneciente al archivo *docker-compose.yml*.

Cabe destacar que, para que exista conectividad entre todos los contenedores creados, se configura una red denominada grafana haciendo uso del parámetro "networks" (fuera de la etiqueta "services"), y a la que todos los contenedores pertenecen.

```
version: "3.7"
services:
  grafana:
    build: ./build/grafana
    container_name: grafana-dashboard
    restart: always
    networks:
      grafana:
        aliases:
          - grafana-dashboard
    ports:
      - 80:3000
    volumes:
      - grafana_data:/var/lib/grafana
    environment:
      ## DATABASE ##
      - GF_DATABASE_TYPE=mysql
      - GF_DATABASE_HOST=name
      - GF_DATABASE_NAME=name
      - GF_DATABASE_USER=change_me_pls
      - GF_DATABASE_PASSWORD=change_me_pls
      ## ANALYTICS ##
      - GF_ANALYTICS_REPORTING_ENABLED=false
      ## SECURITY ##
      - GF_SECURITY_ADMIN_USER=change_me_pls
      - GF_SECURITY_ADMIN_PASSWORD=change_me_pls
      ## SNAPSHOTS ##
      - GF_SNAPSHOTS_EXTERNAL_ENABLED=false
      ## USERS ##
      - GF_USERS_ALLOW_SIGN_UP=false
      - GF_USERS_ALLOW_ORG_CREATE=false

    prometheus:
        build: ./build/prometheus-server
        restart: always
        networks:
          grafana:
            aliases:
              - grafana-prometheus-server
```

Listing 4.21: Servicio de Grafana y Prometheus de docker-compose.yml

Para finalizar, se prueban las nuevas modificaciones levantando de nuevo la infraestructura y aprovisionando la instancia de Grafana con dichas modificaciones aplicadas. Luego se accede al servidor de Grafana vía el dominio web y se comprueban las métricas obtenidas.

```
database:
    image: mysql:latest
    container_name: grafana-database
    restart: always
    networks:
      grafana:
        aliases:
          - grafana-database
    ports:
      - 5000:3306
    volumes:
      - grafana_database_data:/var/lib/mysql
    environment:
      - MYSQL_DATABASE=name
      - MYSQL_USER=name
      - MYSQL_PASSWORD=change_me_pls
      - MYSQL_RANDOM_ROOT_PASSWORD=yes
```

Listing 4.22: Servicio de Database del docker-compose.yml

Capítulo 5

Diseño e implementación del CLI

En este capítulo se explica el diseño y la implementación del CLI para la gestión de las operaciones que se van a realizar sobre la infraestructura de proyectos creada en el proveedor cloud AWS. El objetivo principal del CLI es crear diferentes instancias en el entorno de desarrollo seleccionado para uno o varios proyectos en un entorno cloud. Actualmente, solo soporta el proveedor AWS, pero se ha implementado de una manera que resulte sencilla la incorporación de un nuevo proveedor.

Las secciones del presente capítulo se corresponden con los Sprints 3 y 4 (véase Tabla 3.2), exceptuando la primera que se corresponde con una breve explicación del CLI existente, y la última donde se realiza la estimación y el coste total de todos los Sprints realizados en el TFG. Además, al finalizar los Sprints inicialmente planificados en tiempo y forma, se decidió añadir dos adicionales (Sprints 5 y 6) ya que se consideró que se disponía de tiempo suficiente. En el Sprint 5 se genera una base datos DynamoDB para almacenar la información concreta de un proyecto, mientras que en el Sprint 6 se implementa la creación de instancias para un proyecto, contemplando en que subred se levanta y si es pública o privada. También se modifica la base de datos para almacenar el CIDR de cada subred y conocer cuales están libres u ocupadas para un proyecto. De esta manera, se ha modificado el Product Backlog añadiendo las nuevas tareas. En cada sección se muestra una tabla detallada del Sprint correspondiente con todas sus tareas así como su esfuerzo y coste.

5.1 Estudio del estado actual del CLI

Al iniciar el TFG, de igual manera que sucedía con la infraestructura, ya existía una primera versión del CLI. Antes de entrar en detalle sobre sus funcionalidades, se explica como es la estructura típica de las aplicaciones CLI implementadas en lenguaje Go.

Habitualmente, en los lenguajes de programación existe una estructura de directorios que se suele seguir en la realización de un proyecto, como puede ser el caso de Java. En Go no es

diferente y existen guías de recomendación de cómo se debe estructurar una aplicación CLI, donde básicamente se debe crear una estructura modular separando el código de la aplicación del código de entrada a la aplicación. Tanto en esta primera versión como en la final se utiliza la siguiente estructura:

- /cmd: Carpeta que contiene el punto de entrada a la aplicación definido en el *main.go*.
- /configs: Esta carpeta contiene un archivo de configuración JSON donde se le indican diferentes parámetros que se utilizan al configurar el CLI.
- /internal/app: Esta carpeta contiene todos los paquetes que representan los componentes de la aplicación que no son expuestos y que no tienen por qué funcionar fuera de la misma.
- /pkg: Carpeta que contiene todos aquellos componentes de la aplicación que se quieren exponer y que deben funcionar fuera de la misma. En la versión inicial existían:
  - /api: API REST para para clonar repositorios Git.
  - /cli: Contiene los archivos que generan cada comando que tiene el CLI. En esta versión existen tres comandos que permiten ejecutar: 1) *terraform apply*; 2) *terraform destroy*; y 3) *git clone*. Todos ellos fueron creados a partir de la biblioteca externa Cobra [35].
  - /config: Define la configuración de la aplicación (nivel de log, variables de entorno para usar los comando de Terraform, etc) a partir de la biblioteca Viper [36].
  - /git: Transforma el comando *git clone* para que sea utilizable por la aplicación usando las funciones de */shell*.
  - /interactive: Utilizando la biblioteca Promtui [37] se crea un menú interactivo para poder seleccionar en que entorno (VPC) se desea realizar una acción.
  - /shell: Conjunto de funcionalidades para poder ejecutar los comandos externos como comandos propios de la aplicación. Para ello se utiliza la biblioteca Exec [38], entre otras.
  - /terraform: Transforma los comandos de Terraform para que sean utilizables por la aplicación usando las funciones de */shell*.
- /vendor: Esta carpeta contiene todas las dependencias con biblioteca externas.

Para gestionar las dependencias entre las carpetas de la propia aplicación o las bibliotecas externas, Go utiliza sus denominados módulos. Estos se definen en el archivo *go.mod*, donde se indica el path del módulo y su versión. Estas dependencias se almacenan en la carpeta */vendor*.

En esta versión inicial del CLI, a parte de lo existente en */configs* y */pkg*, dentro de*/cmd* está el archivo *main.go* que ejecuta la aplicación llamando a */internal/app/main.go* para arrancar el CLI y llamando a */pkg/config* para seleccionar la configuración asociada. En */internal/app* se encuentra el archivo de arranque del CLI y el archivo que configura el sistema de logs de la aplicación utilizando la biblioteca externa Logrus [39].

Resumiendo, en la versión inicial se permite clonar directorios de Git, que son almacenados en un carpeta concreta existente en el directorio del proyecto. El objetivo es clonar un directorio que contenga archivos de Terraform para levantar instancias en la infraestructura creada. Una vez clonado el directorio, se puede ejecutar el comando "up" donde a través del menú interactivo seleccionamos el entorno (VPC) donde se desea levantar el número de máquinas que se indiquen en los archivos de Terraform descargados. Posteriormente, existe el comando "down" para eliminar todas las instancias creadas. Previamente, en el archivo JSON de */configs* se deben proporcionar las claves AWS necesarias para trabajar con Terraform y la carpeta donde se almacenan las descargas de Git.

5.2 Sprint 3 - Diseño del CLI

En esta sección se explica el diseño del nuevo CLI a partir de las tareas obtenidas tras la reunión de planificación del Sprint 3, que se muestran en la Figura 5.1.

Sprint	Id tarea	Nombre de la tarea	Inicio	Fin	Esfuerzo	Coste (€)
3	1	Iniciación al lenguaje de Programación Go	10/11/2020	11/11/2020	4	92,52
3	2	Adaptación al estado actual del CLI	12/11/2020	19/11/2020	10	291,3
3	3	Diseño del nuevo CLI	23/11/2020	23/11/2020	2	46,26
				Total	16	430,08

Figura 5.1: Planificación y estimación del Sprint 3

5.2.1 Tarea 1- Iniciación a Go

Durante esta primera tarea se lleva a cabo una iniciación al lenguaje de programación Go mediante la realización de diversos tutoriales y lectura de su documentación. Sin embargo, antes de continuar hay que aclarar como se ejecuta el CLI. Este tipo de aplicaciones Go se compilan con el comando *go build*, aunque antes de ello, se debe compilar el archivo *go.mod* con el comando *go mod vendor* para cargar las dependencias. Posteriormente, se ejecuta *go build /cmd/main.go* que genera el binario con el que ejecutar el CLI y, dependiendo del sistema operativo, se generará un binario diferente. Durante el desarrollo de este TFG se ha compilado a través del CMD de Windows, obteniendo el binario *cli.exe*, y a través del WSL, obteniendo el binario *cli*.

5.2.2 Tarea 2 - Adaptación al estado actual del CLI

Después de haber adquiridos los conocimientos básicos, se inicia la Tarea 2 donde se dedica un tiempo a estudiar el estado actual del CLI. A continuación se listan algunas de las bibliotecas utilizadas, acompañadas de una breve definición:

- La biblioteca Cobra proporciona una interfaz simple para crear interfaces CLI, basada en una estructura de comandos, argumentos e indicadores (flags), donde los comandos representan acciones, los argumentos son opciones y los indicadores son modificadores de esas acciones. Más concretamente, este CLI utiliza una estructura multiparte que debe especificarse en este orden:

  1. La llamada base al programa *cli.exe*.
  2. El comando que especifica la operación a realizar junto a sus opciones o parámetros si son necesarios (flags).
  3. Opciones o parámetros (flags) generales del CLI necesarios por la operación. Esta opción podría ir después de la llamada base al programa en vez de al final.

  A continuación se muestra un ejemplo de la estructura:

  cli.exe <command [options/parameters of the command]> [options/parameters]

  Se puede obtener ayuda para cualquier comando o sobre el propio CLI escribiendo como parámetro *help*, *–help* o *-h*.

- La biblioteca Viper se utiliza para configurar aplicaciones en Go. Algunas de sus funcionalidades son: lectura de configuraciones a partir de archivos, lectura de variables de entorno o lectura de indicadores de línea de comandos.

- Logrus es la biblioteca de gestión de logs con la que es posible configurar hasta siete niveles de log, aunque para el CLI solo se contemplan cinco. Ordenados de menor a mayor nivel son: “debug” para establecer información para depurar, “info” para notificar algo relevante, “warn” para establecer las advertencias, “error” para notificar los errores, “fatal” notifica el error y cierra la aplicación.

- La biblioteca Promtui permite crear menús de selección interactivos para CLIs.

Además, para el correcto funcionamiento se necesita tener instalado Terraform y el cliente de AWS en la máquina donde se ejecute.

5.2.3 Tarea 3 - Nuevo diseño del CLI

Una vez estudiado el código y la estructura de la primera versión del CLI se decide hacer algunas modificaciones. Por un lado, se decide que todo el código que ejecuta la aplicación se

localice en el archivo *main.go* de la carpeta */cmd*, fusionando así los archivos */cmd/main.go* y */internal/app/main.go*. Por otro lado, se modifica la estructura interna de la carpeta */pkg*, eliminando */pkg/api*, */pkg/git* y el archivo correspondiente a su comando en */pkg/cli*. Se considera que la funcionalidad del API de Git es innecesaria, debido a que para el objetivo de creación de las instancias con definir una carpeta dentro del directorio del proyecto con los archivos de Terraform es suficiente, así se consigue una aplicación más simple solo con las funcionalidades verdaderamente necesarias.

Como uno de los objetivos es añadir un comando que ejecute Packer, se crea dentro de */pkg* la carpeta */modules*, donde dentro de ella se añade */terraform* y */packer*, que contendrá lo necesario para transformar los comandos de Packer y que sean utilizables por la aplicación. Por último, con vistas a incorporar más funcionalidades en un futuro, se crea la carpeta */pkg/utils* donde se introducen los archivos de */shell*. A continuación se muestra el resultado final del directorio */pkg*:

- /cli
- /config
- /interactive
- /modules
- /utils

5.3 Sprint 4 - Implementación del CLI y sus comandos

En esta sección se explica la implementación del nuevo CLI a partir de las tareas obtenidas tras la reunión de planificación del Sprint 4, que se muestran en la Figura 5.2.

Sprint	Id tarea	Nombre de la tarea	Inicio	Fin	Esfuerzo	Coste (€)
4	1	Refactorización del código existente, eliminando dependencias innecesarias para los nuevos objetivos decidios en el diseño	24/11/2020	26/11/2020	6	108,78
4	2	Implentación del comando capaz de arrancar una o varias instancias en una VPC	30/11/2020	03/12/2020	8	185,04
4	3	Implementación del comando capaz de eliminar todas las instancias de una VPC	07/12/2020	07/12/2020	2	46,26
4	4	Implentación del comando capaz de crear una AMI personalizada en AWS	08/12/2020	09/12/2020	3	79,39
				Total	19	419,47

Figura 5.2: Planificación y estimación del Sprint 4

5.3.1 Tarea 1 - Refactorización del código para el nuevo CLI

Una vez obtenido el nuevo diseño se aplican las refactorizaciones de código acordadas. Se elimina del CLI todo el código referente a Git, se añade el directorio */pkg/modules*, el */utils* y se modifica el archivo */cmd/main.go* para que ejecute la aplicación sin tener que llamar a */internal/app/main.go*. Al eliminar la dependencia de Git, se decide crear el directorio *./cache* donde se almacenan los archivos de Terraform y de Packer, que utilizaran sus comandos para realizar sus funciones. Para indicar dentro de cada clase que se trabaje contra ese directorio, dentro del archivo JSON de */configs* se establece como directorio actual de trabajo, y gracias a la biblioteca Viper se puede referenciar en cualquier parte del proyecto. Así, cuando se desee ejecutar el código de Terraform bastaría con usar las funciones de Viper para obtener el directorio.

Como se explicó en la Sección 5.2.2, el CLI se implementa a partir de la biblioteca Cobra. En este momento existe un parámetro general (*–environment|-e*) que identifica el entorno (VPC) que se está utilizando, donde actualmente solo está soportado el entorno cloud AWS y el entorno local, que hace referencia a VirtualBox para la realización de las operaciones de algunos comandos. En la implementación de los comandos se contempla este flag con un switch de modo que para la opción *–environment local* se defina un implementación y para *–environment aws* otra. El fragmento de código 5.1 muestra la creación de lo que sería la entidad CLI y cómo añadir comandos a partir del constructor proporcionado por Cobra. Con la variable "rootCmd" se llama a la biblioteca de Cobra para crear un comando, en este caso el comando inicial, es decir, lo que se obtiene tras ejecutar el binario sin añadir ningún parámetro. La etiqueta "use" define el nombre del comando, "short" y "long" son para mostrar una descripción breve o más extensa cuando se quiera obtener ayuda y "RunE" implementa la funcionalidad del comando. En este caso como es para la ejecución solamente del binario, llama a "cmd.Help()" para mostrar la descripción del CLI. La función "init()" es un método específico de Go que establece una porción de código que debe ejecutarse antes que cualquier otra parte de su paquete y aquí se utiliza para crear el flag de entornos anteriormente explicado. Por último, se observa la función "Run(Config interface)" la cual se invoca desde el archivo *main.go* para arrancar el CLI. Nótese que la inicial del nombre de la función está con mayúsculas, y esto en Go significa que esa función puede ser llamada desde una clase de otro paquete.

Para el caso de añadir un comando se sigue la misma distribución que con "rootCmd" y su flag, pero añadiendo en la función "init()" la llamada a ese nuevo comando (*rootCmd.AddCommand(var_newCommand)*). En la estructura de la aplicación dentro de */pkg/cli* se encuentra un archivo *main.go* que contiene el código anteriormente mostrado y un archivo *.go* por cada comando existente.

```
var rootCmd = &cobra.Command{
  Use:   "cli",
  Short: "Pequeña definición",
  Long: "Explicación más extensa",
  RunE: func(cmd *cobra.Command, args []string) error {
    if len(args) == 0 {
      cmd.Help()
      return errors.New("No arguments provided, leaving")
    }
    return nil
  },
}
func init() {
  rootCmd.PersistentFlags().StringVarP(&env, "environment", "e",
    "local", "Project env (local, aws)")
}
func Run(Config interface{}) error {
  rootCmd.SilenceUsage = true
  err := rootCmd.Execute()
  if err != nil {
    return err
  }
  return nil
}
```

Listing 5.1: Código Go para la creación de comandos con la biblioteca Cobra

5.3.2 Tarea 2 - Comando de arranque de instancias

En esta tarea se modifica el comando existente para poder crear una o varias instancias en el entorno (VPC) seleccionado de la infraestructura. Dicho comando se denomina "up" y dispone de dos flags obligatorios: 1) *–repository|-r*, que se usa para indicar el directorio que contiene los archivos Terraform; y 2) *–name|-n*, para indicar el nombre del proyecto. Cabe mencionar que este comando solo contempla la implementación para el entorno AWS (*–environment aws*), siendo su objetivo principal ejecutar *terraform apply* sobre unos archivos Terraform seleccionados para crear una o varias instancias en un entorno de la infraestructura.

El primer paso para su implementación es añadir al directorio de trabajo *./cache* una subcarpeta */terraform* que contendrá a su vez una carpeta (*/terraform-instances*) con los archivos de Terraform necesarios para crear las instancias. La carpeta */terraform-instances* contiene tres subdirectorios (*/dev*, */pre*, */pro*) con los archivos correspondientes para cada VPC (no se contempla la posibilidad de crear instancias para la VPC de administración). Estos archivos contienen el recurso *aws_subnet* para crear la subred pública donde se levantan las instancias y el módulo para crear una instancia a partir de la AMI personalizada. Además, cada uno contiene un bloque backend para indicar el bucket S3 donde se almacena el fichero de estado Terraform correspondiente, debido a que se desea que múltiples personas puedan ejecutar el CLI al mismo tiempo y no tener problemas con los ficheros de estado guardados en local.

A continuación se detallan los pasos del funcionamiento del comando "up" (ejemplo de uso: *cli.exe up -r terraform-instances -e aws*):

1. Se selecciona como directorio de trabajo la ruta obtenida de concatenar a *./cache/terraform/* el valor indicado con el flag *–repository*.

2. Se selecciona el nombre del proyecto obtenido con el flag *–name*.

3. A través de un menú de selección se obtiene el nombre del entorno donde se quieren crear las instancias. Este menú se encuentra dentro de un bucle por lo que permite crear instancias en distintos entornos en una única ejecución del comando.

4. Se crea la ruta donde se almacena el fichero de estado en el bucket. Esta ruta se crea concatenando el nombre del proyecto obtenido en el paso 2 y el nombre del entorno obtenido en el paso 3. De este modo se formaría la ruta con el siguiente formato: */{nombre-proyecto}/{nombre-vpc}-instances/terraform.state*

5. Se realiza el *terraform init -key="ruta"* pasándole como parámetro la ruta donde se almacenará el fichero de estado en el bucket.

6. Se pide por pantalla en número de instancias a crear.

7. Se realiza el *terraform apply -var instance_count="nº"* pasándole como parámetro el el número de instancias indicado, que se corresponde con una variable de los archivos de Terraform.

8. Se crean adecuadamente las instancias determinadas.

En el caso de querer crear más instancias en un entorno, cuando se pide por parámetro el número de instancias se deberá introducir el número total de instancias (es decir, las que ya existen más las que se quieren crear). Esto es debido a que en el estado de Terraform donde se almacenan los datos de los elementos ya existentes, cuando se realiza una modificación primero se mira los elementos que hay y luego se añaden los nuevos.

La implementación de los parámetros que admiten los comandos *terraform init* y *terraform apply* se realiza dentro de */pkg/modules/terraform*, de modo que Terraform recibe: *terraform init -key="ruta"* y *terraform apply -var instance_count="nº"*.

5.3.3 Tarea 3 - Comando de eliminación de instancias

En esta tarea se crea el comando opuesto a "up", el cual para un entorno (VPC) seleccionado de la infraestructura elimina todas sus instancias. Este comando, denominado "down", también dispone del flag *–repository* obligatorio, usado para indicar el directorio que contiene

los archivos Terraform, y el flag *–name* obligatorio, usado para indicar el nombre del proyecto. Al igual que "up", solo se considera la implementación para el entorno AWS.

A continuación se detallan los pasos del funcionamiento del comando "down" (ejemplo de uso: *cli.exe down -r terraform-instances -e aws*):

1. Se selecciona como directorio de trabajo la ruta obtenida de concatenar a *./cache/terraform/* el valor indicado con el flag *–repository*.
2. Se selecciona el nombre del proyecto obtenido con el flag *–name*.
3. A través de un menú de selección se obtiene el nombre del entorno donde se quieren eliminar las instancias. Este menú se encuentra dentro de un bucle por lo que permite eliminar todas las instancias de varios entornos en una ejecución del comando.
4. Se obtiene la ruta donde se almacena el fichero de estado en el bucket. Esta ruta se crea concatenando el nombre del proyecto obtenido en el paso 2 y el nombre del entorno obtenido en el paso 3. De este modo se formaría la ruta con el siguiente formato: */{nombre-proyecto}/{nombre-vpc}-instances/terraform.state*.
5. Se realiza el *terraform init -key="ruta"* pasándole como parámetro la ruta donde se almacena el fichero de estado en el bucket.
6. Se realiza el *terraform destroy* para eliminar todos los elementos que existen según el fichero de estado proporcionado en el *terraform init*.
7. Se eliminan adecuadamente las instancias determinadas.

5.3.4 Tarea 4 - Comando de construcción de una AMI

En esta tarea se implementa el comando para construir una AMI a partir de Packer y todos los elementos relacionados. Por un lado, y del mismo modo que con Terraform, hay que añadir en el JSON de */configs* la configuración necesaria para que reconozca la instalación de Packer de nuestra máquina. Por otro lado, en */pkg/modules* se añade la carpeta */packer* que contiene la información necesaria para ejecutar el comando *packer build* que construye la AMI. Por último, se crea el comando que llama a */pkg/modules/packer* para ejecutar el comando *packer build* sobre unos templates de Packer. Los archivos que usa el comando de Packer se almacenan en el directorio *./cache/packer* del mismo modo que se hace con los de Terraform, y su contenido es el mismo que el explicado en la Sección 4.2.2.

El comando se crea en */pkg/cli* y se denomina "image", contemplándose la implementación para el entorno local (*–environment local*) y para AWS (*–environment aws*). Soporta un flag, *–architecture*, para indicar el tipo del sistema operativo (Ubuntu, Debian, etc) que se corresponde con el directorio que contiene los archivos de Packer. El objetivo principal del comando

es crear el OVF de la máquina virtual de Virtualbox para el entorno local, y crear la AMI personalizada para el entorno AWS.

A continuación se detallan los pasos del funcionamiento del comando "image" para ambos entornos (ejemplo de uso: *cli.exe image –architecture ubuntu -e local/aws*):

- Entorno local:

  1. Se selecciona como directorio de trabajo la ruta obtenida de concatenar a *./cache/packer/packer-ami-base-* el valor indicado con el flag *–architecture*.
  2. Se ejecuta el comando *packer build* indicándole como parámetro *-only=virtualbox-iso*.
  3. Se obtiene como resultado el OVF para crear la máquina virtual en VirtualBox.

- Entorno AWS:

  1. Se selecciona como directorio de trabajo la ruta obtenida de concatenar a *./cache/packer/packer-ami-base-* el valor indicado con el flag *–architecture*.
  2. Se ejecuta el comando *packer build* indicándole como parámetro *-only=amazon-ebs*.
  3. Se construye la AMI personalizada en AWS.

5.4 Nuevo product backlog

Tras finalizar el Sprint 4 (Sección 5.3), y tras determinar que se disponía de suficiente tiempo hasta la fecha de entrega del TFG, el Product Owner decidió aumentar el Product Backlog con nuevos requisitos que surgieron a lo largo del proceso de desarrollo. En la Tabla 5.1 se muestra el nuevo Product Backlog.

La implementación de las nuevas tareas se ha dividido en dos Sprints: el Sprint 5 que realiza las tareas 9 y 10, y el Sprint 6 que realiza las tareas 11, 12 y 13. En el Sprint 5 se diseña e implementa una base de datos para almacenar la información de los proyectos. En el Sprint 6 se modifican los comandos "up" y "down" para que contemplen la posibilidad de crear o destruir varias subredes (públicas o privadas) en un mismo entorno, y que se almacene en la base de datos la información de las subredes que cada proyecto está utilizando. También se crea una nueva opción para acceder a un CLI interactivo. En la Tabla 5.2 se puede observar la estimación de los nuevos Sprints.

Id	Descripción funcionalidad	Prioridad	Preestimación del esfuerzo
1	Diseño de la infraestructura de entornos para AWS	Muy alta	4
2	Implementación de la infraestructura en AWS siguiendo el paradigma IaC	Muy alta	28
3	Diseño de sistema de monitorización de la infraestructura en AWS	Muy alta	4
4	Implementación de sistema de monitorización de la infraestructura en AWS	Muy alta	6
5	Diseño del CLI para realizar operaciones sobre la infraestructura en AWS	Muy alta	14
6	Implementación del CLI para poder crear comandos que realicen las operaciones deseadas sobre la infraestructura en AWS	Muy alta	16
7	Un usuario levanta una o varias instancias para un proyecto concreto en el entorno seleccionado de la infraestructura en AWS a través del CLI	Muy alta	8
8	Un usuario levanta una AMI propia en AWS a través del CLI	Alta	4
9	Diseño de una base de datos para almacenar la información de los proyectos	Muy Alta	2
10	Implementación de las operaciones CRUD de la base de datos para almacenar la información de los proyectos	Muy Alta	18
11	Un usuario levanta una o varias instancias en una subred (pública o privada) determinada y almacena la información en la base de datos	Muy Alta	12
12	Un usuario destruye una o varias instancias de una subred (pública o privada) determinada y elimina la información de la base de datos	Muy Alta	4
13	Un usuario accede a un CLI interactivo donde ejecutar cualquier comando	Alta	6

Tabla 5.1: Nuevo Product Backlog

Sprint	Id elemento del Product Backlog	Comienzo	Fin	Esfuerzo estimado
Sprint 5	9, 10	10/12/2020	05/01/2021	20
Sprint 6	11, 12, 13	07/01/2020	27/01/2020	22

Tabla 5.2: Estimación de los nuevos Sprints

5.5 Sprint 5 - Diseño e implementación de la base de datos

En esta sección se explica el diseño e implementación de una base de datos DynamoDB en AWS para almacenar la información referente a los proyectos. La Figura 5.3 desglosa el Sprint 5 con sus tareas correspondientes.

Sprint	Id tarea	Nombre de la tarea	Inicio	Fin	Esfuerzo	Coste (€)
5	1	Diseño de la base de datos DynamoDB	10/12/2020	10/12/2020	2	46,26
5	2	Implementación de la base de datos	14/12/2020	16/12/2021	6	108,78
5	3	Implementación del comando para gestionar la base de datos implementada	17/12/2021	21/12/2021	4	92,52
5	4	Modificar el comando de arranque de instancias para que almacene la información de los proyectos en la base de datos y el de eleminación para que la borre	22/12/2021	23/12/2021	4	92,52
5	5	Realización de pruebas unitarias de la capa modelo	04/01/2021	05/01/2021	4	72,52
				Total	20	412,6

Figura 5.3: Planificación y estimación del Sprint 5

5.5.1 Tarea 1 - Diseño de la base de datos DynamoDB

A la hora de diseñar la base de datos que almacene la información de los proyectos, lo primero que hay que decidir es el servicio AWS a usar de entre todos los disponibles (MySQL, DynamoDB, MongDB, etc), que podemos dividir en dos grandes grupos: bases de datos relacionales o no relacionales. En una base de datos relacional se cumplen los principios ACID (Atomicity, Consistency, Isolation, Durability) por lo que presenta un esquema rígido donde su modelo de almacenamiento se basa en tablas con columnas y filas fijas. Mientras que una base no relacional (NoSQL) es más flexible, con un modelo de almacenamiento de documentos (JSON) o basado en pares clave-valor, entre otras posibilidades. Como la base de datos que se quiere utilizar en este TFG es de pequeño tamaño y se desea almacenar en la nube usando documentos JSON, se opta por implementar una base de datos NoSQL, escogiendo el servicio DynamoDB de entre los disponibles, debido a su bajo coste y a la escalabilidad que ofrece.

Para diseñar la base de datos se crea la entidad Project para definir el modelo de dominio. Project será una estructura de datos en Go formada por: 1) la clave primaria "ProjectID" que define el identificador de un proyecto; 2) "ProjectCode" que define el código de un proyecto; 3) "Name" para especificar su nombre; 4) "Sector" para determinar su sector; y 5) "Responsible" para asociar un responsable a un proyecto.

Posteriormente, se utilizan distintos patrones de diseño para lograr un desacoplamiento entre la infraestructura y la base de datos. Para independizar la lógica de negocio de la capa de acceso a datos se aplica el patrón repositorio. El uso de este patrón separa el modelo de dominio de los datos de la capa de persistencia, y de este modo se consigue abstracción con respecto a la base de datos concreta que se está utilizando. Para ello, se crea una interfaz repositorio que define los métodos CRUD que se implementan en una clase concreta para una base de datos determinada (DynamoDB en este caso) y para conectarse con DynamoDB se crea una sesión con AWS mediante el patrón singleton (instancia única) limitando así la creación de sesiones. Posteriormente, se aplica el patrón fachada para implementar la capa de la lógica de negocio, cuyo objetivo es estructurar un entorno de programación y reducir así su complejidad con la división en subsistemas minimizando las comunicaciones y dependencias entre ellos. Se compone por una interfaz para que los comandos del CLI puedan acceder a los casos de uso del modelo del dominio y una clase que implementa dicha interfaz. En la Figura 5.4 se muestra el diagrama del diseño de la base de datos con sus clases concretas.

5.5.2 Tarea 2 - Implementación de la base de datos DynamoDB

Para la implementación de la base de datos DynamoDB se utiliza el SDK de Go proporcionado por AWS.

El diseño de la base de datos DynamoDB se implementa dentro de la estructura del pro-

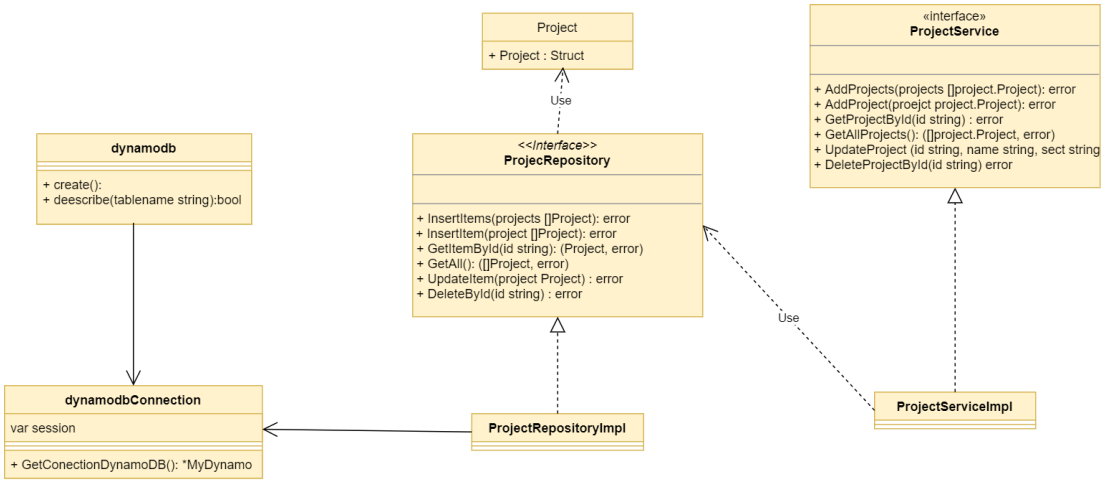


Figura 5.4: Diseño de la base de datos DynamoDB

yecto en el directorio */pkg/database*, que contiene la clase *dynamodb.go* donde se implementan los métodos para crear la tabla de proyectos y para comprobar la existencia de una tabla ("describe()"), y la clase *dynamodbConnection.go* donde se crea la sesión para conectarse con DynamoDB. El CLI no implementa ninguna funcionalidad para crear la tabla de proyectos, si no que en *main.go* se añade en la función "init()" que mediante el método "describe()" compruebe si la tabla existe y si no la cree.

Para el resto de elementos de la base de datos se crea el directorio */pkg/model* que está compuesto por tres subcarpetas: */entity*, */repository* y */service*. En */entity* se encuentra la clase *project.go* que contiene la estructura que define los elementos de la base de datos. En */repository* se implementa el patrón repositorio donde se encuentra la interfaz que define los métodos CRUD en *projectRepository.go* y la implementación concreta usando el SDK de Go de AWS en la clase *projectRepositoryImpl.go* para DynamoDB. Dentro de esta clase se obtiene un objeto de la sesión y se implementan los métodos de la interfaz. Seguidamente, en */service* se implementa el patrón fachada, donde primero se define la interfaz en *projectService.go* cuyos métodos se invocarán desde los comandos definidos en */pkg/cli* , y segundo se implementan los métodos concretos que llaman al repositorio para ejecutar sus funciones contra la base de datos.

Cabe mencionar que en Go no existen las excepciones de la misma manera que en otros lenguajes como Java. Para crear una excepción personalizada y que una función pueda devolver un error, se compara el valor obtenido con nulo y en caso de no ser nulo se devuelve ese error. Dicho error se almacena en una variable y se gestiona con el sistema de logs para que aparezca por pantalla. Además, para validar el formato de los datos que se ingresan en DynamoDB se utiliza un validador mediante la biblioteca Validator [40].

5.5.3 Tarea 3 - Comando para trabajar contra DynamoDB

Una vez implementada la base de datos y las funciones CRUD para trabajar con ella, se crea el comando "db" que llama a los métodos de */pkg/model/service* para crear, obtener, eliminar o actualizar proyectos. Este comando se crea del mismo modo que los vistos anteriormente, añadiendo una nueva clase en */pkg/cli* donde se define el comando a partir de la biblioteca Cobra. En el directorio *./cache* se añade la carpeta */json* debido a que como DynamoDB es una base de datos no relacional se pueden cargar datos a partir de un fichero en formato JSON. La ejecución de este comando se corresponde con: *cli.exe db [flags] -e aws*.

Las funcionalidades que contempla este comando "db", a través de sus parámetros (flags), son las siguientes:

- *–insertItems*: inserta elementos en la tabla proyectos a partir de un archivo JSON que se debe encontrar en *.cache/json/*.
- *–insertItem*: inserta un nuevo elemento en la tabla de proyectos introduciendo los campos como entrada.
- *–listIteml*: obtiene un elemento de la base de datos de proyectos por su identificador y lo muestra.
- *–update|-u*: actualiza un elemento de la base de datos de proyectos.
- *–deleteItem|-d*: elimina un elemento de la base de datos de proyectos.

5.5.4 Tarea 4 - Modificación de los comandos "up" y "down"

Hasta este momento, el flag *–name* permitía indicar el nombre del proyecto para un determinado entorno. Como ahora los proyectos se almacenan en una base de datos, se decide eliminar este flag de ambos comandos y obtener el valor correspondiente de la base de datos. Para ello, se genera un menú interactivo que lista los proyectos existentes en la base de datos, almacenando en una variable el identificador del proyecto. De esta manera, la nueva ruta que se le pasa al comando *terraform init -key* para almacenar el estado en el bucket S3 sería: *{ProjecID}/{nombre-vpc}-instances/terraform.state*.

5.5.5 Tarea 5 - Realización de pruebas unitarias

Para probar el correcto funcionamiento de la base de datos se realiza un test de pruebas unitarias, creado en */pkg/model/repository/projectRepositoryImpl_test.go*. En Go los test siempre llevan la etiqueta "_test" y se crean en el paquete donde se encuentra el código que van a probar, debido a que el comando *go test* va a buscar los archivos terminados con esa extensión y así poder ejecutarlos. Para crear el test, se recurre a la biblioteca Dynamock [41] para crear

mocks y poder probar las interacciones con la base de datos DynamoDB. También se utiliza la biblioteca Assert [42] para obtener un conjunto de herramientas que faciliten la realización de los tests. En la Figura 5.5 se muestra el resultado obtenido.

Figura 5.5: Cobertura de los test unitarios

5.6 Sprint 6 - Gestión de subredes y CLI interactivo

En este Sprint se implementa la creación y eliminación de subredes, públicas o privadas, para los entornos de un proyecto concreto y un comando para ejecutar el CLI de forma interactiva. En la Figura 5.6 se detallan las tareas desglosadas del Sprint.

Sprint	Id tarea	Nombre de la tarea	Inicio	Fin	Esfuerzo	Coste (€)
6	1	Modificación de la entidad Proyecto de la base de datos para almacenar la asignación de subredes (CIDR y pública o privada) a las instancias de un proyecto concreto	07/01/2021	07/01/2021	1	53,3
6	2	Implementación de los métodos necesarios para la correcta asignación del CIDR de cada subred para cada proyecto	07/01/2021	12/01/2021	5	105,65
6	3	Implementación de los métodos para insertar y borrar en la base de datos el campo que referencia a las subredes de un entorno en un proyecto	13/01/2021	14/01/2021	2	46,26
6	4	Modificación del comando de arranque de las instancias para contemplar la asignación de la subred y añadir esta información a la base de datos	18/01/2021	20/01/2021	6	108,78
6	5	Modificación del comando de eliminación de las instancias por contemplar la asignación de la subred y eliminar la información de la base de datos	20/01/2021	21/01/2021	2	46,26
6	6	Implementación de un comando para ejecutar el CLI de manera interactiva	25/01/2021	27/01/2021	6	108,78
				Total	22	469,03

Figura 5.6: Planificación y estimación del Sprint 6

5.6.1 Tarea 1 - Modificación de DynamoDB para la asignación de subredes

Para mantener un registro de qué subred está asignada para cada entorno de cada proyecto, lo primero que se hace es modificar la entidad proyecto de DynamoDB, añadiéndole un nuevo campo "Environment" que es una lista de tipo "Env". Esta nueva estructura está formada por tres campos: "Name" (string), "PublicSubnet" y "PrivateSubnet" (ambos son listas de strings). La finalidad de esto es almacenar para un proyecto concreto sus tres posibles entornos, donde "Name" es el nombre del entorno y las listas almacenan la parte del bloque CIDR que identifica la subred pública o privada. En la Figura 5.7 se observa esta entidad.

```
type Env struct {
    Name          string   `validate:"required"`
    PublicSubnet  []string `validate:"required"`
    PrivateSubnet []string `validate:"required"`
}

type Project struct {
    ProjectID   int    `validate:"required"`
    ProjectCode string `validate:"required"`
    Name        string `validate:"required"`
    Sector      string `validate:"required"`
    Responsible string `validate:"required"`
    Enviroments []Env
}
```

Figura 5.7: Entidad proyecto

5.6.2 Tarea 2 - Implementación de la asignación de subredes

Una vez definido como se almacena esta información en la base de datos hay que establecer como se asignan las subredes. Como se explica en el Capítulo 4, cada VPC presenta un CIDR 10.X.0.0/16, siendo 'X' el dígito que representa el entorno (0 para mng, 1 para dev, 2 para pre y 3 para pro). Teniendo eso en cuenta, la máscara de subred escogida debe ser mayor que /16 y se necesita identificar de algún modo al proyecto dentro del CIDR. Para ello, se reserva la tercera parte del CIDR, asignándole como valor el identificador del proyecto. Para tener un número razonable de subredes por proyecto se decide utilizar una máscara /27 que permite tener hasta ocho subredes (0/27, 32/27, 64/27, 96/27, 128/27, 160/27, 192/27, 224/27). Las cuatro primeras se dedican para subredes públicas y las cuatro restantes para subredes privadas. De este modo, se tiene un CIDR 10.X.P.S/27 para cada subred de un proyecto donde 'X' identifica al entorno, 'P' al proyecto y 'S' a la subred (pública o privada). Con esta asignación se pueden crear hasta 254 proyectos con 8 subredes posibles para cada entorno, siendo 4 públicas y 4 privadas, de 30 hosts cada una de ellas.

La implementación de esta asignación se realiza en */pkg/subnetting/subnet.go*, y en DynamoDB en las listas de la estructura "Env" solo se almacena el valor que identifica a la subred

(0/27), no el bloque CIDR completo. El diagrama UML de la clase *subnet.go* se muestra en la Figura 5.8, donde "GetFreeCidr" llama a "GenerateProjectCidr" para obtener la parte del CIDR correspondiente al proyecto, y posteriormente llama a "generateSubnetCidr" que a su vez se apoya en "findFreeSubnetInList" para obtener la parte referente a la subred. En la Figura 5.9 se representa en un diagrama de secuencias el proceso descrito.

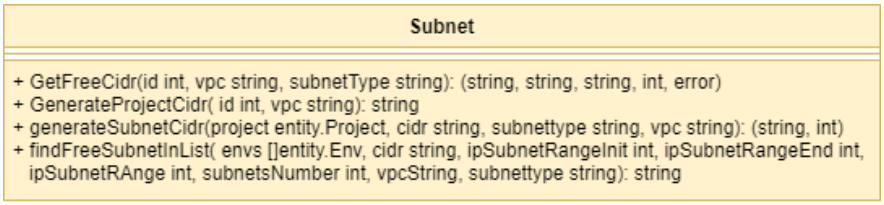


Figura 5.8: Clase Subnet.go

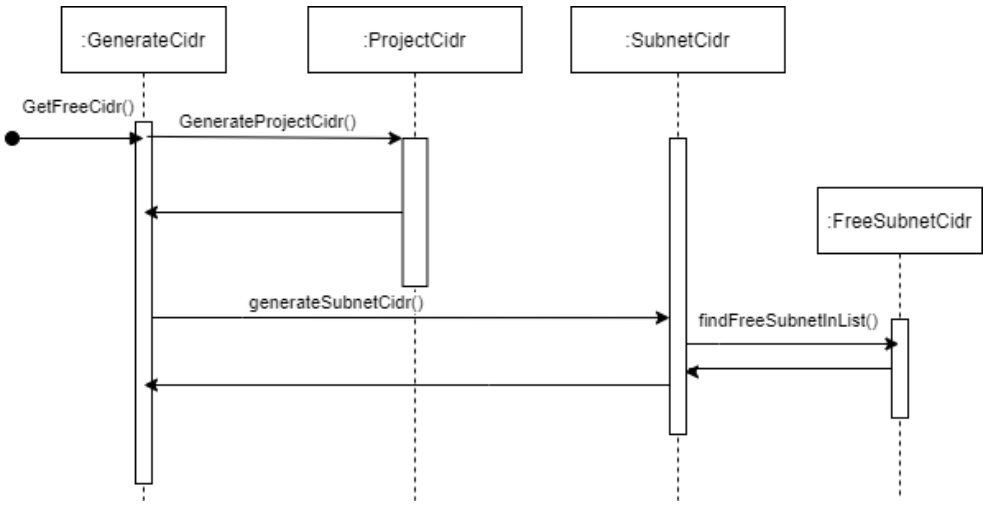


Figura 5.9: Diagrama de secuencia para la obtención de un CIDR

5.6.3 Tarea 3 - Gestión de subredes en la base de datos

En esta tarea se crean los métodos necesarios para almacenar y eliminar las subredes de un entorno de la base de datos. Una vez establecida la asignación del CIDR hay que almacenar la información correspondiente en DynamoDB. Para ello se implementan nuevos métodos en */pkg/model/service* que se utilizan en los comandos "up" y "down cuando se produzca la asignación. En la Figura 5.10 se muestran los nuevos métodos en la interfaz projectService.go.

A continuación se explica la utilidad de cada método nuevo:

- "AddProjectSubnet": Con este método se añade a un proyecto concreto el CIDR para el tipo de subred especificado en el entorno seleccionado.

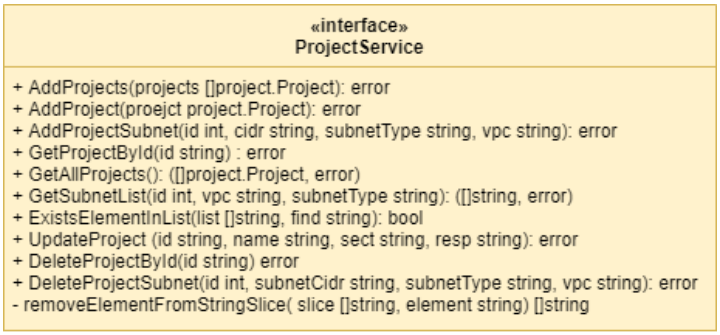


Figura 5.10: Interfaz con los nuevos métodos para la gestión de subredes

- "GetSubnetList": Recorre la lista de entornos ([]Env) para obtener la lista de subredes públicas o privadas del entorno especificado para un proyecto concreto.
- "ExistsElementInList": Devuelve un booleano indicando si un elemento existe o no en una lista.
- "DeleteProjectSubnet": Elimina el valor de subred pasado por parámetro de la lista de subredes que se especifique para un entorno de un proyecto concreto.
- "removeElementFromStringSlice": Método auxiliar que utiliza "DeleteProjectSubnet" para eliminar la subred de la lista.

5.6.4 Tarea 4 - Modificación del comando "up"

Una vez implementados los métodos para gestionar las subredes con DynamoDB, se modifica el comado "up" para que establezca esta funcionalidad. Antes de nada, se modifica el valor de la ruta (key) del bucket S3 para añadirle la subred, quedando *{ProjecID}/{nombre-vpc}-instances/subred{CIDR}/terraform.state*, y se crea un nuevo menú de selección para escoger si las instancias que se desean crear van a estar en una subred pública o privada.

Para crear una instancia en una subred concreta, se modifica el comando *terraform apply* para que admita como variable el bloque CIDR, de esta manera el comando quedaría de la siguiente forma: *terraform apply -var CIDR="10.0.0.0/27"*. En cuanto a la obtención del CIDR, se llama al método "GetFreeSubnet" de */pkg/subnetting/subnet.go* que se encarga de asignar una subred libre para el proyecto en cuestión.

Después de la ejecución del comando Terraform, se guarda el valor de la subred en la base de datos. También se implementa la opción de crear instancias para una subred que ya está en uso. Además, se añade una nueva funcionalidad para eliminar del directorio *./cache/terraform* los archivos de estado de Terraform que se guardan en local, para evitar posibles problemas de compatibilidad con los archivos existentes en el bucket S3.

A continuación se detallan los pasos del funcionamiento del comando "up":

1. Se selecciona como directorio de trabajo la ruta obtenida de concatenar a *./cache/terraform/* el valor indicado con el flag *–repository*.
2. A través de un menú de selección se obtiene el proyecto donde se quieren crear las instancias.
3. A través de un menú de selección se obtiene el nombre del entorno donde se quiere crear las instancias. Este menú se encuentra dentro de un bucle por lo que permite crear instancias en distintos entornos en una ejecución del comando.
4. A través de un menú de selección se determina si se quiere crear las instancias en una subred pública o privada.
5. En caso de que ya exista una subred asignada al entorno del proyecto, se pregunta al usuario si la instancia que se va a crear se quiere en esa subred o en otra nueva. La nueva subred es la que se obtiene de ejecutar el método "GetFreeSubnet".
6. Se pide por pantalla en número de instancias a crear para la nueva subred o la subred ya existente.
7. Se realiza el *terraform init* pasándole como parámetro la ruta donde se almacenará el fichero de estado en el bucket. Esta ruta se obtiene con el nombre del proyecto, del entorno y el bloque de subred que se obtuvieron en los pasos 2, 3 y 5.
8. Se realiza el *terraform apply -var instance_count="nº" CIDR="10.0.0.0/27" type="public-/private"* pasándole como parámetros el bloque CIDR, si es pública o privada y el número de instancias indicado, que se corresponden con las variables de los archivos de Terraform.
9. Se crean adecuadamente las instancias determinadas.
10. Se almacena el valor de la subred del bloque CIDR en la base de datos para el entorno y el proyecto seleccionados, en caso de no haber escogido una subred existente.
11. Se elimina el estado de Terraform que se almacena de forma local.

5.6.5 Tarea 4 - Modificación del comando "down"

Del mismo modo que el comando "up", también se realiza la misma modificación para el comando "down" pero añadiendo un nuevo flag (*–deleteInstances*) que permite indicar el número de instancias que se desean eliminar de una subred. Como el comando *terraform*

destroy elimina todos los elementos que contenga el fichero de estado, se eliminarían todas las instancias de una subred por la forma en que se almacena el bucket S3. Por ello se crea ese nuevo indicador para poder eliminar un número determinado de instancias de una subred.

A continuación se detallan los pasos del funcionamiento de este comando "down" (ejemplo de uso: *cli.exe down -r terraform-instances [–deleteInstances] -e aws*):

1. Se selecciona como directorio de trabajo la ruta obtenida de concatenar a *./cache/terraform/* el valor indicado con el flag *–repository* .
2. A través de un menú de selección se obtiene el proyecto donde se quieren eliminar las instancias.
3. A través de un menú de selección se obtiene el nombre del entorno donde se quiere eliminar las instancias de una subred.
4. A través de un menú de selección se determina si se quiere eliminar las instancias de una subred pública o privada.
5. Se obtiene de la base de datos la lista de subredes del proyecto indicado, para el entorno proporcionado para la subred indicada.
6. Se realiza el *terraform init* pasándole como parámetro la ruta donde almacenará el fichero de estado en el bucket. Esta ruta la obtiene con el nombre del proyecto, del entorno y de la subred que se obtuvieron en los pasos anteriores.
7. En caso de haber seleccionado el flag *–deleteInstances*, se pide por pantalla el número de instancias a eliminar para la subred existente. Se realiza el *terraform apply* pasándole como parámetros el bloque CIDR, si es pública o privada y el número de instancias indicado, que se corresponden con las variables de los archivos de Terraform. En caso contrario, se realiza un *terraform destroy* y se eliminan todas las instancias de la subred.
8. Se eliminan adecuadamente las instancias determinadas.
9. Se borra el valor de la subred de la base de datos para el entorno y el proyecto seleccionados.
10. Se elimina el estado de Terraform que se almacena de forma local.

5.6.6 Tarea 6 - Comando CLI interactivo

En esta tarea se crea el comando "ishell" que implementa un shell interactivo para el CLI utilizando la biblioteca Ishell [43], que permite crear comandos pero, a diferencia de Cobra, dentro de un shell que se ejecuta de forma interactiva. Para su implementación se crea

un archivo en */pkg/cli* donde con el método "ishell.New()" se crea el shell interactivo y con "AddCmd()" se añaden los comandos existentes en el CLI al mismo. Un ejemplo de la sintaxis del comando sería: *cli.exe -e aws ishell.*

5.7 Estimaciones y coste total

A lo largo del Capítulo 4 y del capítulo actual se ha detallado el trabajo y el coste para cada Sprint de forma individual. En la Figura 5.11 se muestra el resultado de unificar las planificaciones de todos los Sprints llevados a cabo en el TFG.

Sprint	Inicio	Fin	Esfuerzo	Coste (€)
Sprint 1	05/10/2020	29/10/2020	32	920,16
Sprint 2	02/11/2020	09/10/2020	10	181,3
Sprint 3	10/11/2020	23/11/2020	16	430,08
Sprint 4	24/11/2020	09/12/2020	19	419,47
Sprint 5	10/12/2020	05/01/2021	20	412,6
Sprint 6	07/01/2021	27/01/2021	22	469,03
		Total	119	2832,64

Figura 5.11: Resumen de la planificación de los Sprints

La duración total del desarrollo de la infraestructura y del CLI es de aproximadamente unos 4 meses dedicando 20 horas semanales, comenzando a finales de Septiembre y finalizando a finales de Enero. Esto se traduce en 119 puntos de esfuerzo que equivalen a 297,5 horas al cambio. Sin embargo, una vez finalizado el desarrollo del CLI se realizó también un manual de usuario (véase Anexo B), cuyo esfuerzo no se refleja en la Figura 5.11. Además, tampoco se ha incluido el tiempo necesario para la redacción y revisión de este documento.

Teniendo eso en cuenta, la duración total del TFG asciende a 5 meses aproximadamente, donde el coste ha sido estimado a partir de un salario de un desarrollador becado de 5,25€/hora y para los tutores de 40€/hora. Para el coste económico total de cada tarea de los Sprints se tiene en cuenta la aportación del Development Team y del Product Owner cuando corresponda. Debido a ello, aunque en algunas tareas se necesite el mismo esfuerzo, el coste puede variar en función del tiempo aportado por el tutor profesional de la empresa. También, en la tarea de redacción de la memoria se incluye en el coste la aportación de los tutores dedicada a su revisión.

En cuanto al coste de la infraestructura, también hay que contemplar los gastos producidos por utilizar servicios de AWS, considerando que la infraestructura no está todo el tiempo levantada, puesto que durante el proceso de desarrollo se crea y se destruye según convenga. En la Tabla 5.3 se muestran los costes de los principales recursos para la región de AWS

us-east-2, donde además se añade el coste de 0,023 USD por GB/mes del bucket S3. De esta manera se obtiene que el gasto total de la infraestructura, dependiendo del número de instancias levantadas en cada VPC, siendo 0,7421 USD por hora, teniendo en cuenta el coste del espacio utilizado del bucket S3 que en el estado actual del proyecto no supera un GB, y también que AWS solamente cobra por IPs elásticas cuando no están en uso.

Recurso	Coste por hora
Instancia EC2	0,6516 USD
ELB	0,0225 USD
IP Elástica	0,005 USD
NAT Gateway	0,045 USD

Tabla 5.3: Coste por hora de los principales recursos de AWS

Cabe mencionar que no es sencillo obtener una estimación del coste total de AWS durante la realización del TFG, pero podemos suponer que de los 5 meses de duración al menos en 4 se estuvo trabajando con la infraestructura levantada intermitentemente. Si suponemos que estuvo levantada aproximadamente 8 horas semanales durante 4 meses con al menos 5 instancias se obtiene un gasto estimado de 431.836€. En resumen, en la Figura 5.12 se muestra la totalidad del esfuerzo y coste directo para este TFG y en la Figura 5.13 el coste total, teniendo en cuenta el gasto estimado producido por el uso de AWS. Finalmente, también se muestra en forma de diagrama de Gantt la planificación del TFG en la Figura 5.14.

Tarea	Esfuerzo	Coste (€)
Desarrollo de la infraestructura y el CLI	297,5 horas	2832,64
Manual del CLI	10 horas	52,5
Redacción de la memoria	85 horas	793,75
Total	392,5 horas	3678,89

Figura 5.12: Resumen del esfuerzo y el coste total del TFG

Ámbito	Coste (€)
Desarrollo TFG	3678,89
AWS	431,836
Total	4110,726

Figura 5.13: Resumen del coste total teniendo en cuenta AWS

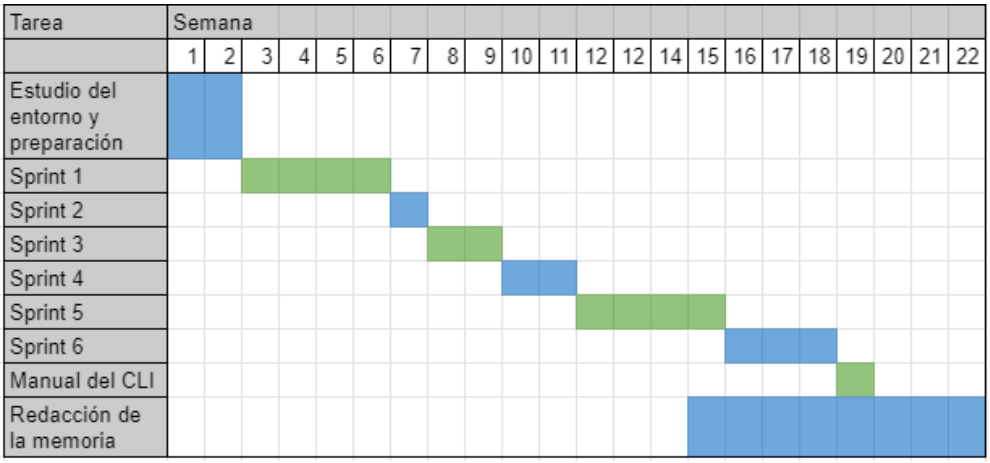


Figura 5.14: Diagrama de Gantt

5.8 Pruebas

En esta sección se hace una breve reflexión sobre las pruebas realizadas a lo largo del desarrollo del proyecto sobre el funcionamiento de todos los componentes, tanto para la infraestructura como para el CLI.

Para la infraestructura se han realizado pruebas funcionales, puesto que es lo más habitual cuando se trata de testear los elementos creados con Terraform. Para el CLI también se han realizado mayormente pruebas funcionales, exceptuando las pruebas unitarias realizadas contra la base de datos DynamoDB (véase la Sección 5.5.5).

Terraform no dispone de funcionalidad propia para crear tests, sino que se apoya en bibliotecas externas. Una de las más utilizadas es Terratest [44], la cual a partir de archivos Go permite utilizar comandos de Terraform para probar sus funcionalidades. La idea de uso de esta biblioteca es levantar una infraestructura, ejecutar los tests y destruir la infraestructura. Sin embargo, como este proceso es relativamente costoso, se optó por realizar pruebas funcionales, es decir, según se van creando los distintos elementos se comprueba a través de la interfaz web de AWS si su estado es el correcto. Este tipo de pruebas se realizaron tanto en el Sprint 1 como en el Sprint 2 para comprobar todos los componentes de la infraestructura y su correcto funcionamiento.

Para probar el funcionamiento de los comandos del CLI, también se realizaron pruebas funcionales a lo largo de sus respectivos Sprints, comprobando todos los posibles casos de uso. Solamente se realizaron test automatizados para la base de datos a partir de las pruebas unitarias.

Capítulo 6

Conclusiones y líneas futuras

Último capítulo de la memoria del TFG donde se comentan las principales conclusiones obtenidas, las lecciones aprendidas y las posibles líneas futuras a seguir.

6.1 Conclusiones

Al inicio de este proyecto se habían fijado unos objetivos que se debían alcanzar, y tras su finalización, se puede afirmar que se ha cumplido con todas las expectativas.

- Se ha diseñado una infraestructura en AWS, altamente escalable, capaz de dar soporte a diferentes entornos y aplicaciones.
- Se ha conseguido implementar esta infraestructura de una manera automatizada gracias a la utilización del paradigma IaC con la herramienta Terraform.
- Con el uso de Packer se ha podido desarrollar imágenes máquina idénticas para las instancias, configurándoles Prometheus para permitir crear una plataforma de monitorización centralizada de la infraestructura a la que es posible acceder a través de un dominio web, que da acceso al servidor de Grafana.
- Se ha desarrollado un CLI en lenguaje Go para gestionar las operaciones de los proyectos y las instancias asociadas a sus entornos.

En cuanto a las lecciones aprendidas, en primer lugar, este proyecto me ha permitido profundizar en el terreno de la infraestructura en cloud de la mano de AWS, que es una temática que me parece muy interesante y muy necesaria en la situación actual. Aunque es cierto que en un par de asignaturas de la mención de Tecnologías de la Información se estudia el uso de este servicio, en ninguna de ellas se sigue el paradigma IaC para automatizar la creación de los elementos.

En segundo lugar, he aprendido a crear AMIs para AWS a través de Packer, la automatización del aprovisionamiento de distintos componentes con Ansible, a utilizar Prometheus para la obtención de series de datos temporales y a usar Grafana para poder mostrar de una forma más visual lo obtenido con Prometheus.

En tercer lugar, he aprendido el lenguaje de programación Go que actualmente es el más utilizado para la realización de CLIs. Dentro de la implementación del CLI, a la hora de estructurar el código me han servido de gran ayuda las asignaturas en las que he trabajado con patrones de diseño, sobre todo para el diseño de la base de datos. También fue la primera vez que utilicé una base de datos no relacional como DynamoDB.

Por último, la elección de este proyecto en el marco de unas prácticas de empresa extracurriculares creo que fue una gran oportunidad para introducirme en el mundo laboral y conocer el proceso de desarrollo de un proyecto en un ámbito real.

6.2 Líneas futuras

El siguiente paso a seguir en el proyecto podría ser la creación de plantillas de instancias para los entornos, es decir, preconfigurar instancias con los elementos habituales más demandadas para los proyectos como bases de datos, proxies o balanceadores. De esta manera, se aumentaría la velocidad de creación de los entornos para los proyectos.

En lo referente al CLI, por un lado tenemos que está diseñado de manera que se puedan crear nuevas funcionalidades para operar contra la infraestructura a medida que vayan surgiendo. Sin embargo, el siguiente paso sería transformar el CLI en una aplicación web, esto permitiría no tener que instalar nada y poder acceder desde cualquier sitio de una forma prácticamente instantánea, además de aprovechar un interfaz gráfico para mejorar la experiencia de usuario frente al uso de la línea de comandos.

Por otro lado, en el estado actual del CLI y teniendo en cuenta las dependencias necesarias para su utilización, creo que sería conveniente "dockerizar" el CLI. De este modo, se tendría un contenedor Docker con todo lo necesario para ejecutar el CLI correctamente.

Por último, en cuanto al servicio de monitorización, Grafana permite crear alertas sobre las métricas por lo que podría ser interesante configurar alguna en función de lo que se necesite.

Apéndices

Apéndice A

Archivos de configuración de Packer

```
{
    "variables": {
        "version": "0.0.1",
        "disk_size": "20480",
        "cpus": "1",
        "memory": "1024",
        "os_version": "18.04.2",
        "os_arch": "amd64",
        "os_type": "server",
        "ami_virtualization_type": "hvm",
        "delete_ebs_volumes": "true",
        "aws_volume_size": "8",
        "aws_os_version": "18.04",
        "aws_region": "us-east-2",
        "aws_instance_type": "t2.micro"
    },
  "provisioners": [
        {
            "type": "file",
            "only": [
        "amazon-ebs"
      ],
            "source": "./conf/defaults.cfg",
            "destination": "/tmp/defaults.cfg"
        },
        {
            "type": "shell",
            "only": [
        "amazon-ebs"
      ],
            "inline": [
```

```
            "sudo mv /tmp/defaults.cfg
/etc/cloud/cloud.cfg.d/defaults.cfg"
          ]
      },
    {
      "type": "shell",
      "override": {
        "virtualbox-iso": {
          "execute_command": "echo 'everis' | {{.Vars}} sudo -S -E
bash '{{.Path}}'"
        },
        "amazon-ebs": {
          "execute_command": "echo 'everis' | {{.Vars}} sudo -S -E
bash '{{.Path}}'"
        }
      },
      "script": "scripts/common/ansible.sh"
    },
    {
      "type": "ansible-local",
      "playbook_file": "ansible/main.yml",
      "galaxy_file": "ansible/requirements.yml"
    },
    {
      "type": "shell",
      "override": {
        "virtualbox-iso": {
          "execute_command": "echo 'everis' | {{.Vars}} sudo -S -E
bash '{{.Path}}'"
        },
        "amazon-ebs": {
          "execute_command": "echo 'everis' | {{.Vars}} sudo -S -E
bash '{{.Path}}'"
        }
      },
      "script": "scripts/common/cleanup.sh"
    }
    ],
    "builders": [
    {
      "type": "virtualbox-iso",
      "boot_command": [
        "<esc><wait>",
        "<esc><wait>",
        "<enter><wait>",
        "/install/vmlinuz<wait>",
```

```
        " auto<wait>",
        " console-setup/ask_detect=false<wait>",
        " console-setup/layoutcode=es<wait>",
        " console-setup/modelcode=pc105<wait>",
        " debconf/frontend=noninteractive<wait>",
        " debian-installer=es_ES<wait>",
        " fb=false<wait>",
        " initrd=/install/initrd.gz<wait>",
        " kbd-chooser/method=es<wait>",
        " keyboard-configuration/layout=ES<wait>",
        " keyboard-configuration/variant=ES<wait>",
        " locale=es_ES<wait>",
        " netcfg/get_domain=vm<wait>",
        " netcfg/get_hostname=everis<wait>",
        " grub-installer/bootdev=/dev/sda<wait>",
        " noapic<wait>",
        " preseed/url=http://{{ .HTTPIP }}:{{ .HTTPPort
    }}/preseed.cfg<wait>",
        " -- <wait>",
        "<enter><wait>"
      ],
      "boot_wait": "10s",
      "disk_size": "{{ user `disk_size` }}",
      "guest_os_type": "Ubuntu_64",
      "headless": true,
      "http_directory": "http",
      "iso_urls": [
        "iso/ubuntu-{{user `os_version`}}-{{user `os_type`}}-{{user
    `os_arch`}}.iso",
        "http://releases.ubuntu.com/{{user
    `os_version`}}/ubuntu-{{user `os_version`}}-{{user
    `os_type`}}-{{user `os_arch`}}.iso"
      ],
      "iso_checksum":
    "file:http://old-releases.ubuntu.com/releases/18.04.2/SHA256SUMS",
      "ssh_username": "everis",
      "ssh_password": "everis",
      "ssh_port": 22,
      "ssh_wait_timeout": "15000s",
      "shutdown_command": "echo 'everis'|sudo -S shutdown -P now",
      "guest_additions_path": "VBoxGuestAdditions_{{.Version}}.iso",
      "virtualbox_version_file": ".vbox_version",
      "vm_name": "packer-ubuntu-{{user `os_type`}}-{{ user
    `os_version` }}-{{ user `os_arch` }}",
      "vboxmanage": [
        [
```

```
113           "modifyvm",
114           "{{.Name}}",
115           "--memory",
116           "{{ user `memory`}}"
117         ],
118         [
119           "modifyvm",
120           "{{.Name}}",
121           "--cpus",
122           "{{ user `cpus` }}"
123         ]
124       ]
125         },
126         {
127       "type": "amazon-ebs",
128       "access_key": "",
129       "secret_key": "",
130       "ami_name": "ubuntu-{{ user `os_type`}}-{{ user `os_version`
    }}-{{ user `os_arch` }}",
131       "instance_type": "{{ user `aws_instance_type` }}",
132       "region": "{{ user `aws_region` }}",
133       "ssh_username": "ubuntu",
134       "source_ami_filter": {
135         "filters": {
136                     "name": "ubuntu/images/hvm-ssd/ubuntu-*-{{user
    `aws_os_version`}}-{{user `os_arch`}}-server-*",
137           "root-device-type": "ebs"
138         },
139         "most_recent": true,
140         "owners": [""]
141       },
142       "ami_block_device_mappings": [
143         {
144           "delete_on_termination": "{{ user `delete_ebs_volumes`}}",
145           "device_name": "/dev/sda1",
146           "volume_size": "{{ user `aws_volume_size` }}",
147           "volume_type": "gp2"
148         }
149       ],
150       "ami_description": "Auto-generated ubuntu {{ user `os_type`
    }} AMI. OS architecture: {{ user `os_arch` }}. OS version: {{
    user `os_version` }}",
151       "ami_virtualization_type": "{{ user
    `ami_virtualization_type`}}",
152       "force_delete_snapshot": true,
153            "force_deregister": true,
```

```
            "user_data_file": "./conf/defaults.cfg"
    }
  ]
}
```

Listing A.1: Template de la AMI

```
apt -y update && apt-get -y upgrade
apt-get install software-properties-common
apt-add-repository universe
apt-get -y update
apt -y install python-pip python-dev
pip install ansible
```

Listing A.2: Script de instalación de Ansible

```
---
- name: Configure server
  hosts: localhost
  become: yes
  gather_facts: yes
  vars:
    ####################################
    ## PROMETHEUS NODE EXPORTER ROLE ##
    ####################################
    prometheus_node_exporter_version: 0.18.1
    ####################
    ## SECURITY ROLE ##
    ####################
    security_ssh_port: 22
    security_ssh_password_authentication: "no"
    security_ssh_permit_root_login: "no"
    security_ssh_usedns: "no"
    security_sudoers_passwordless:
    - everis
    security_sudoers_passworded: []
    security_autoupdate_enabled: true
    security_autoupdate_blacklist: []
    security_autoupdate_reboot: false
    security_autoupdate_reboot_time: "03:00"
    security_autoupdate_mail_to: ""
    security_autoupdate_mail_on_error: false
    security_fail2ban_enabled: true
    ####################
    ## FIREWALL ROLE ##
    ####################
    firewall_state: started
```

```
    firewall_enabled_at_boot: true
    firewall_allowed_tcp_ports:
    - "22"
    - "80"
    - "443"
    - "9100"
    firewall_allowed_udp_ports: []
    firewall_forwarded_tcp_ports: []
    # - { src: "22", dest: "2222" }
    # - { src: "80", dest: "8080" }
    firewall_forwarded_udp_ports: []
    firewall_additional_rules: []
    firewall_ip6_additional_rules: []
    firewall_log_dropped_packets: true
    firewall_disable_firewalld: true
    firewall_disable_ufw: false
    ############################
    ## USER MANAGEMENT TASKS ##
    ############################
    packages: ['curl', 'wget', 'nano', 'devscripts', 'debhelper',
    'build-essential', 'dh-make', 'network-manager', 'unzip' ,
    'mlocate', 'software-properties-common', 'dos2unix', 'python']
tasks:
- name: Install useful packages
    apt:
      name: "{{ packages }}"
      state: latest

 roles:
    - geerlingguy.java
    - geerlingguy.security
    - geerlingguy.firewall
    - undergreen.prometheus-node-exporter
```

Listing A.3: Playbook para aprovisionar la imagen creada con Packer

Apéndice B

Guía de usuario del CLI

LEPIN Command Line Interface

Guía de usuario

Índice

Índice de figuras

1. ¿Que es LEPIN CLI?

LEPIN Command Line Interface (CLI) es una herramienta, implementada en Go, para facilitar y centralizar todas las operaciones que se pueden realizar sobre una infraestructura ya creada en un proveedor cloud. Su principal objetivo es crear y administrar los entornos de varios proyectos, almacenándolos en una base de datos DynamoDb en el proveedor cloud AWS. Lepin CLI interactúa mediante el uso de comandos en el shell proporcionado al ejecutar la aplicación.

Actualmente esta disponible la primera versión, versión 1.0, y se distribuye mediante un fichero comprimido con todo lo necesario para su ejecución. El proveedor cloud actualmente soportado es AWS.

2. Instalación de LEPIN CLI

En este apartado se explica el proceso de instalación de LEPIN Command Line Interface (LEPIN CLI).

2.1. Prerrequisitos

Se necesita tener instalado y configurado en el sistema:

- AWS CLI
  - Download: `https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html`
  - Configure:`https://docs.aws.amazon.com/cli/latest/reference/configure/` -¿Configurar tus claves AWS y establecer la región por defecto us-east-2.
- Terraform
  - Download: `https://www.terraform.io/downloads.html`
  - Install: `https://learn.hashicorp.com/terraform/getting-started/install.html`
- Packer
  - Download: `https://www.packer.io/downloads`

2.2. Instalación

El usuario dispone del ejecutable *lepin.exe* ya listo para su uso dentro del fichero comprimido proporcionado.

3. Mediante LEPIN CLI

En esta sección se presentan las opciones disponibles en el LEPIN Command Line Interface (LEPIN CLI)

3.1. Estructura del comando

En este tema se explica cómo se estructura el comando LEPIN Command Line Interface (LEPIN CLI) y como utilizarlo. LEPIN CLI utiliza una estructura multiparte en la línea de comandos que debe especificarse en este orden:

1. La llamada base al programa de lepin.exe
2. El comando que específica que operación realizar junto a sus opciones o parámetros si son necesarios (flags)
3. Opciones o parámetros (flags) generales de la CLI necesarios por la operación. Esta opción podría ir después de la llamada base al programa en vez de al final.

lepin.exe <command [options and parameter of the command] >[options and parameter]

3.2. Obtener ayuda con el LEPIN CLI

Puedo obtener ayuda para cualquier comando escribiendo *help*, *–help*, *-h*. Por ejemplo el siguiente comando muestra la ayuda de las opciones generales de LEPIN CLI y los comandos (véase Figura 1).

lepin.exe help o *lepin.exe –help* o *lepin.exe -h*

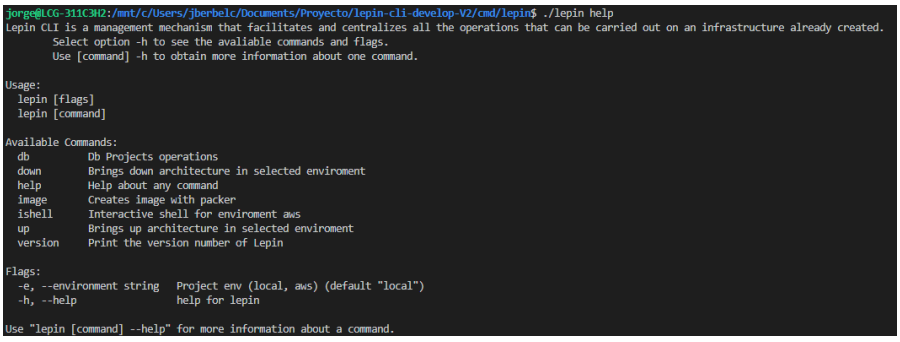


Figura 1: Ayuda del CLI

3.3. Especificación de parámetros generales para LEPIN CLI

Existe un parámetro general *-e, –environment* que identifica que entorno estamos utilizando. Actualmente solo esta soportado el entorno cloud *aws* y el entorno *local*. El entorno local hace referencia a VirtualBox para la realización de las operaciones de algunos comandos. Por defecto el valor de este parámetro es *local*. Un ejemplo de su uso sería:

lepin.exe ¡comando¿-e aws

3.4. Comandos LEPIN CLI

A continuación se muestra una lista con los comandos, su descripción, un ejemplo de su declaración y los parámetros permitidos, soportados por el LEPIN CLI:

- db: Permite realizar operaciones CRUD (Create, Read, Update, Delete) sobre los elementos de la tabla proyectos de DynamoDB (véase Figura 2).
  - *lepin.exe db [flags] [global flags]*
  - Parámetros:
    - *insertItems*: inserta elementos en la tabla proyectos a partir de un archivo .json que se debe encontrar en *.cache/json/*.
    - *insertItem*: inserta un elemento en la tabla de proyectos introduciendo los campos como entrada.
    - *listItem* o *l*: obtiene un elemento de la base de datos de proyectos por su id y lo muestra.
    - *update* o *u*: actualiza un elemento de la base de datos de proyecto.
    - *deleteItem* o *d*: elimina un elemento de la basde de datos de proyectos.

```
jorge@LCG-311C3H2:/mnt/c/Users/jberbelc/Documents/Proyecto/lepin-cli-develop-V2/cmd/lepin$ ./lepin db -h
DynamoDB CRUD operations for manage projects table.

Usage:
  lepin db [flags]

Flags:
  -d, --deleteItem    delete from project database
      --deleteTable   delete BD
  -h, --help          help for db
      --insertItem    insert item in project database from input
      --insertItems   insert items in project database from json in .cache/json/
  -l, --listItem      list item from project database by id
  -u, --update        update item from project database

Global Flags:
  -e, --environment string   Project env (local, aws) (default "local")
```

Figura 2: Ayuda del comando db

- down: Destruye la arquitectura seleccionada para el entorno seleccionado (-e). Si el entorno seleccionado es AWS (-e aws) y el repositorio del proyecto Terraform (-r) es terraform-instances: el comando destruirá la arquitectura creada en un vpc seleccionado de un proyecto concreto. También eliminará los datos correspondientes sobre la información del entorno de AWS (vpc_name, public_subnet_list, private_subnet_list) en la tabla de proyectos de DynamoDB. El valor predeterminado es "local", use -e para editar este valor (véase Figura 3).
    - *lepin.exe down [flags] [global flags]*
    - Parámetros:
        - *repo* o *o*: Indica en nombre del repositorio seleccionado del directorio *.cache/terraform/* que contiene los archivos de Terraform para realizar la operación de *down*.
        - *deleteInstances* o *d*: Elimina las instancias de una subnet (no elimina la subnet). Permite eliminar todas las instancias o de menor a mayor rango en función del tiempo de creación de la instancia. Por ejemplo si existen 3 instancias para un proyecto en un entorno, si introducimos 0 eliminará todas. En cambio si introducimos 2, eliminará la 3 y así sucesivamente. Esto funciona así debido a que esta implementado a través de Terraform (`https://learn.hashicorp.com/tutorials/terraform/count`) y la idea es que el número que se indique es número final de instancias a obtener.

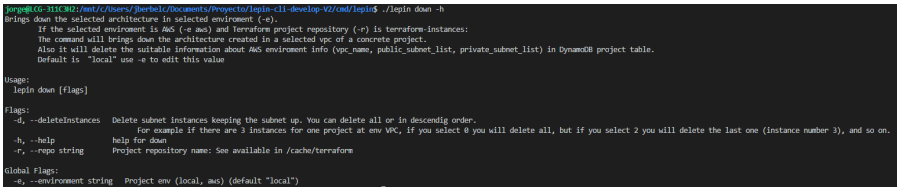


Figura 3: Ayuda del comando down

- image: Crea imágenes en diferentes entornos (local o aws) con diferentes herramientas (virtualbox, amazon-ebs) a través de Packer(véase Figura 4).
    - *lepin.exe image [flags] [global flags]*
    - Parámetros:
        - *builder* o *b*: Constructor de la imagen, 'aws'(amazon ebs) para crear un AMI o 'virtualbox' para crear una máquina virtual en virtualbox en local.
        - *architecture* o *a*: Arquitectura seleccionada para crear la imagen, actualmente solo esta soportada ubuntu 18.04.

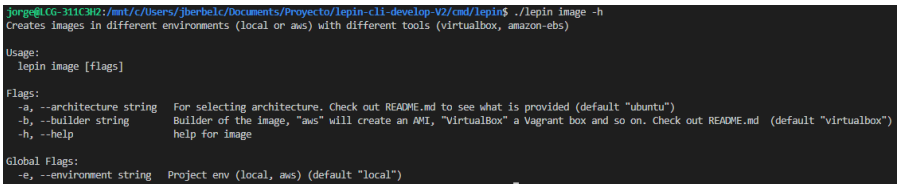


Figura 4: Ayuda del comando image

- ishell: Shell interactivo diseñado para el entorno aws. Los usuarios pueden usar los mismos comandos que normalmente lo hacen con el parámetro -e aws (véase Figura 5).
  - *lepin.exe ishell [flags] [global flags]*

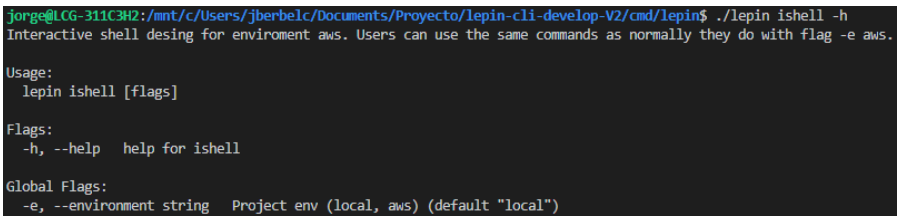


Figura 5: Ayuda del comando ishell

- up: Levanta la arquitectura seleccionada para el entorno seleccionado (-e) a partir de un proyecto Terraform (-r) existente en el repositorio */cache/-terraform*. Si el entorno seleccionado es AWS (-e aws) y el repositorio de proyectos de Terraform (-r) es terraform-instances: el comando creará un número n de instancias en un proyecto existente en una VPC seleccionada con una subred pública o privada asociada. Además, agregará la información del entorno de AWS (vpc_name, public_subnet_list, private_subnet_list) al objeto del proyecto dynamoDB. El valor predeterminado es 'local', use -e para editar este valor (véase Figura 6).
  - *lepin.exe up [flags] [global flags]*
  - Parámetros:
    - *repository* o *r*: Indica en nombre del repositorio seleccionado del directorio *.cache/terraform/* que contiene los archivos de Terraform para realizar la operación de *up*.

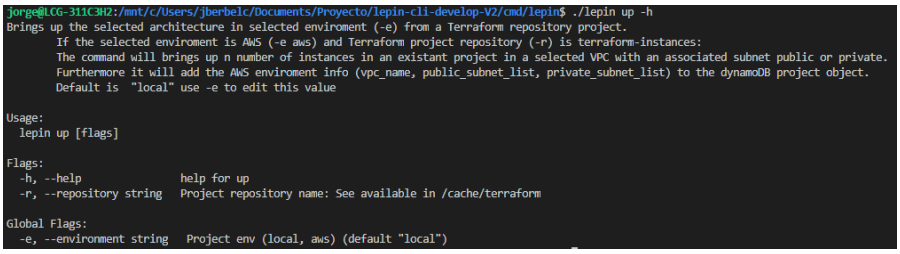


Figura 6: Ayuda del comando up

- version: Imprime por pantalla el número de versión de LEPIN CLI (véase Figura 7).
  - *lepin.exe version [flags] [global flags]*

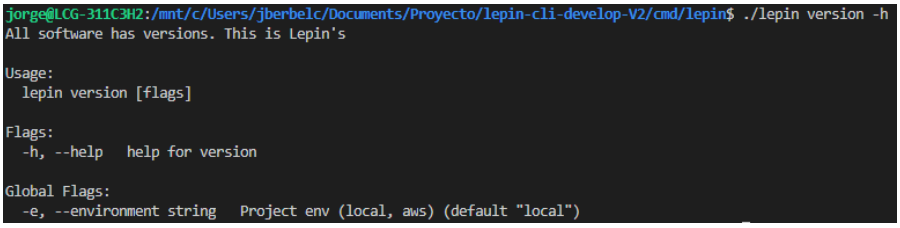


Figura 7: Ayuda del comando version

4. Prueba de funcionamiento de LEPIN CLI

En esta sección se explican los pasos a seguir para levantar o destruir una o varias instancias para un entorno de un proyecto. También se muestra el uso de los comandos para trabajar con la base de datos y el CLI interactivo.

4.1. Paso 1 - Añadir proyectos a la base de datos

El primer paso es utilizar el comando *db* para añadir proyectos a la base de datos DynamoDB (véase Figura 2 para ver los parámetros del comando). Con el comando *lepin .exe -e aws db –insertItems* se carga los proyectos desde un fichero JSON, el nombre del archivo se introduce por pantalla. A continuación se muestra en el fragmento de código 1 el archivo JSON utilizado y el resultado de aplicar este comando en la Figura 8.

Listing 1: Archivo JSON con los proeyctos de ejemplo

```
[
    {
        "projectID" : 1,
```

```
                "ProjectCode" : "EjemploProjectCode1",
                "name" : "Herramienta de desarrollo para ...",
                "sector" : "PS",
                "responsible" : "Persona",
                "environments" : []
        },
        {
                "projectId" : 2,
                "ProjectCode" : "EjemploProjectCode2",
                "name" : "Plataforma de interoperabilidad de la..",
                "sector" : "PS",
                "responsible" : "Persona A",
                "environments" : []
        },
        {
                "projectId" : 3,
                "ProjectCode" : "EjemploProjectCode3",
                "name" : "Plataforma de implemantacion de ...",
                "sector" : "A",
                "responsible" : "Persona",
                "environments" : []
        }
]
```

```
jorge@LCG-311C3H2:/mnt/c/Users/jberbelc/Documents/Proyecto/lepin-cli-develop-V2/cmd/lepin$ ./lepin -e aws db --insertItems
✓ Selected aws environment
Name of the data file (without extension): project_data
Successfully added '1' (EjemploProjectCode1) to table Project
Successfully added '2' (EjemploProjectCode2) to table Project
Successfully added '3' (EjemploProjectCode3) to table Project
```

Figura 8: Comando para insertar proyectos

También se puede añadir un solo proyecto con el comando *lepin .exe -e aws db –insertItem* introduciendo los datos por pantalla (véase Figura 9).

```
jorge@LCG-311C3H2:/mnt/c/Users/jberbelc/Documents/Proyecto/lepin-cli-develop-V2/cmd/lepin$ ./lepin -e aws db --insertItem
✓ Selected aws environment
Select a non existant project id: 4
Select a project Code:EjemploProejctCode4
Select project name: Prueba de proyecto
Select project sector: B
Select project responsible: Persona B
Successfully added '4' (EjemploProejctCode4) to table Project
```

Figura 9: Comando para insertar proyectos

En caso de querer modificar algún parámetro de un proyecto se puede utilizar el comando *lepin .exe -e aws db –update* (véase Figura 10) y para borrar un proyecto se utiliza *lepin .exe -e aws db –deleteItem* (véase Figura 11).

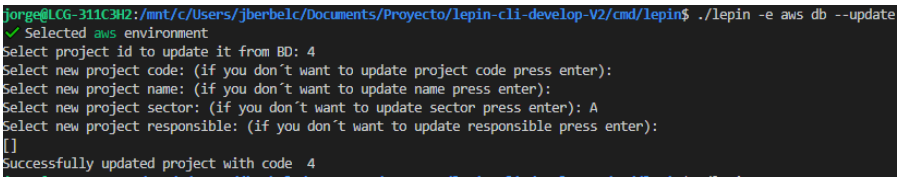


Figura 10: Actualización de un proyecto de la base de datos

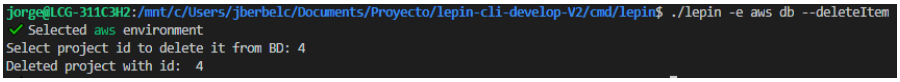


Figura 11: Eliminación de un proyecto de la base de datos

4.2. Paso 2 - Crear AMI personalizada con Packer

Para generar la AMI personalizada para AWS se utiliza el comando *lepin .exe -e image -b aws* que ejecuta el comando *packer build*.

4.3. Paso 3 - Levantar instancias

Para crear instancias en un entorno para un proyecto se utiliza el comando *lepin.exe -e aws up -r terraform-instances*, donde terraform-instances es el directorio donde se encuentran los archivos de Terraform que realizan esta labor. A continuación se muestran las Figuras del funcionamiento de este comando.

1. Primero se selecciona el proyecto (véase Figura 12) y el entorno (véase Figura 13). Se pueden seleccionar más de un entorno a la vez.

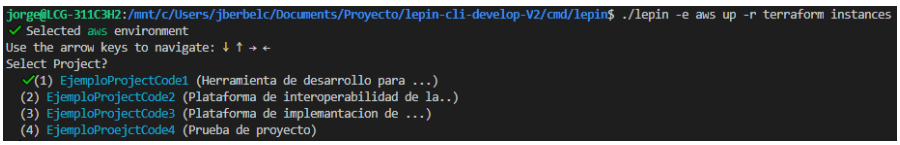


Figura 12: Selección del proyecto

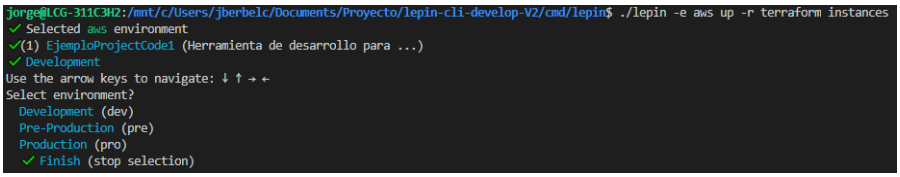


Figura 13: Selección del entorno

2. Se selecciona el tipo se subred (pública o privada) (véase Figura 14).

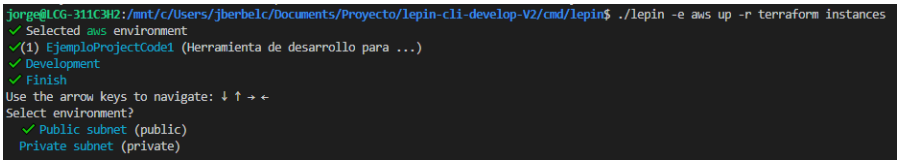


Figura 14: Selección del tipo de subred

3. En caso de que ya exista una subred, se nos pregunta por pantalla si la instancia que vamos a crear la queremos en ese subred o en otra nueva (véase Figura 15).

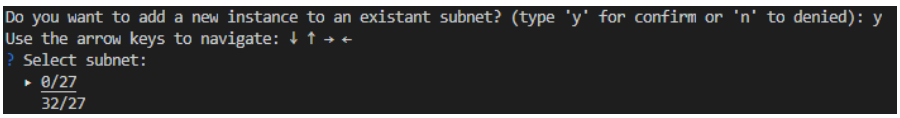


Figura 15: Selección de subred

4. Se indican el número de instancias a crear (véase Figura 16).

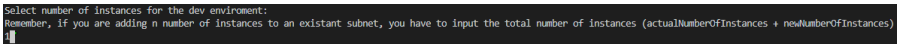


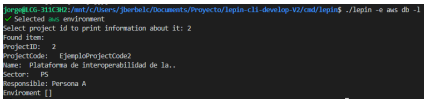
Figura 16: Número de instancias a crear

5. Se ejecutan el *terraform init* y el *terraform apply*

6. Se crea la instancia indicada y se añade la información correspondiente a la base de datos.

Con el comando *lepin .exe -e aws db –listItem* se puede obteber un proyecto y observar sus campos. En la siguiente Figura 17 se muestra un proyecto con subredes asociadas y otro sin ellas.

(a) Proyecto con subredes asociadas



(b) Proyecto sin subredes asociadas

Figura 17: Comparativa entre proyectos con y sin subredes asociadas

4.4. Paso 4 - Eliminar instancias

Para eliminar todas las instancias de una subred se utiliza el comando *lepin.exe -e aws down -r terraform-instances*. Los pasos de ejecución de este comando son similares al del comando "up". Pero llegados a un punto se nos pide indicar la subred de la cual queremos eliminar las instancias, como se muestra en la Figura 18. Una vez finalizada la ejecución se elimina la instancia y se borra la subred de la base de datos.

Figura 18: Comando para eliminar todas las instancias de una subred

En este comando también se puede usar el flag *-d* para eliminar un número determinado de instancias de una subred (*lepin.exe -e aws down -r terraform-instances*). Este caso de uso se muestra en la Figura 19, donde para dos instancias se elimina una de ellas.

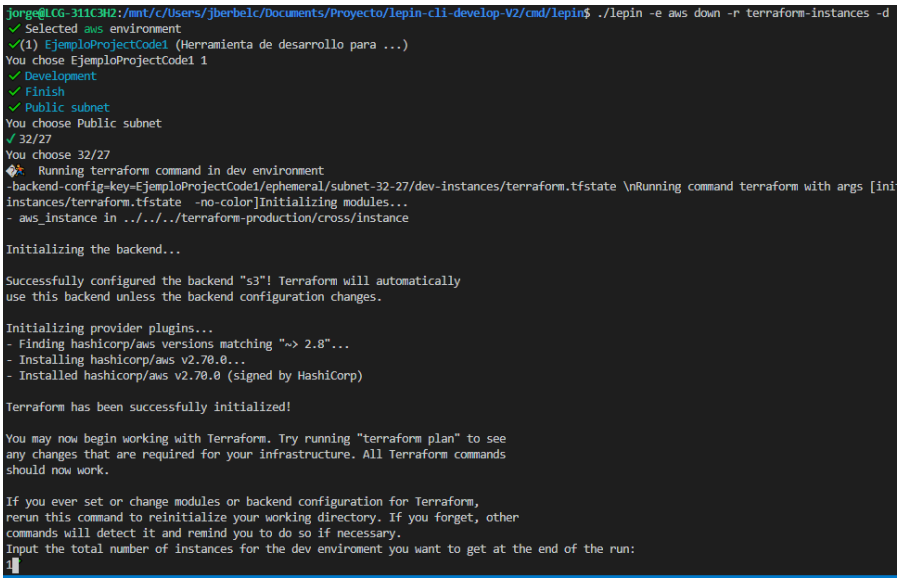


Figura 19: Comando para eliminar una o varias instancias de una subred

4.5. CLI interactivo

Todos los comandos anteriores se pueden ejecutar desde el CLI interactivo utilizando el comando *lepin.exe -e aws ishell*. A continuación se muestran en la Figura 20 la representación de este comando.

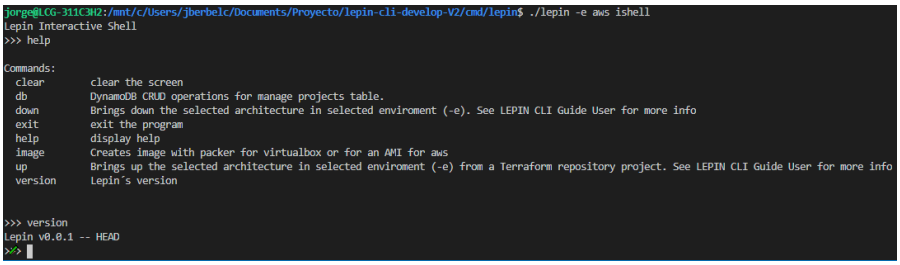


Figura 20: Comando para el CLI interactivo

Lista de acrónimos

ACL Access Control List. 30

AMI Amazon Machine Image. 5

AWS Amazon Web Services. 2

AZ Availability Zones. 5

CLI Command Line Interface. 1

EBS Elastic Block Store. 5

EC2 Elastic Compute Cloud. 5

ELB Elastic Load Balancing. 31

GUI Graphical User Interfac. 13

HCL HashiCorp Configuration Language. 6

HVM Hardware Assisted Virtualization. 30

IaC Infraestructure as Code. 1

IAM (Identity and Access Management. 6

OVF Open Virtualization Format. 8

PromQL Prometheus Query Language. 10

TFG Trabajo Fin de Grado. 1

VPC Virtual Private Cloud. 6

WSL Windows Subsystem for Linux. 14

Bibliografía

[1] V. G. Iglesias and A. Capella, "Entornos para el desarrollo de grandes aplicaciones de gestión de redes," in *Desarrollo de grandes aplicaciones distribuidas sobre internet*, 2005, pp. 45–64.

[2] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, 1st ed. O'Reilly Media, 2015.

[3] "Golang documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://golang.org/

[4] "Web de aws," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://aws.amazon.com/es/

[5] "Web de terraform," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://www.terraform.io/

[6] "Web de packer," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://www.packer.io/

[7] "Web de promtheus," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://prometheus.io/

[8] "Web de grafana," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://grafana.com/

[9] "What is aws?" consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://aws.amazon.com/es/what-is-aws/

[10] "Regiones y zonas de disponibilidad," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://aws.amazon.com/es/about-aws/global-infrastructure/regions_az/

[11] "Documentación de amazon elastic compute cloud," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/AWSEC2/latest/WindowsGuide/concepts.html

[12] "Documentación de amazon simple storage service," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/AmazonS3/latest/dev/Introduction.html

[13] "Documentación de amazon virtual private cloud," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/vpc/latest/userguide/what-is-amazon-vpc.html

[14] "Documentación de amazon dynamo data base," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/amazondynamodb/latest/developerguide/Introduction.html

[15] "Documentación de amazon indetity and access management," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://aws.amazon.com/es/iam/?hp=tile&so-exp=below

[16] "Terraform documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://www.terraform.io/intro/index.html

[17] "Packer documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://www.packer.io/docs

[18] "Ansible documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://docs.ansible.com/ansible/latest/index.html

[19] "Ansible galaxy," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://galaxy.ansible.com/

[20] "Promtheus documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://prometheus.io/docs/introduction/overview/

[21] "Grafana documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://grafana.com/grafana/

[22] "Docker documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://docs.docker.com/get-started/overview/

[23] "What is a virtual machine?" consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://www.redhat.com/es/topics/virtualization/what-is-a-virtual-machine

[24] "Virtualbox documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: http://www.oracle.com/us/technologies/virtualization/oracle-vm-virtualbox-overview-2981353.pdf

[25] "Golang documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://golang.org/doc/

[26] A. A. A. Donovan and B. W.Kernighan, *The Go Programming Language*, 1st ed. Addison-Wesley, 2015.

[27] S. Chacon and B. Straub, *Git Pro*, 2nd ed. Appress, 2014.

[28] "Documentation for visual studio code," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://code.visualstudio.com/docs

[29] "Microsoft docs (faq) wsl," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://docs.microsoft.com/en-us/windows/wsl/faq#:~:text=The%20Windows%20Subsystem%20for%20Linux,desktop%20and%20modern%20store%20apps.

[30] "Documentation for mobaxtream," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://mobaxterm.mobatek.net/

[31] "Manifiesto por el desarrollo Ágil de software," 2001, consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://agilemanifesto.org/iso/es/manifesto.html

[32] J. P. Alexander Menzinsky, Gertrudis López, "Scrum manager: Temario troncal i," 2019, consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://www.scrummanager.net/files/scrum_manager.pdf

[33] K. S. y Jeff Sutherland, "La guía de scrum," 2020, consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://www.scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Spanish-European.pdf

[34] "Documentación de amazon route 53," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://docs.aws.amazon.com/es_es/Route53/latest/DeveloperGuide/Welcome.html

[35] "Library cobra documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://github.com/spf13/cobra

[36] "Library viper documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://github.com/spf13/viper

[37] "Library promtui documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://github.com/manifoldco/promptui

[38] "Library exec documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://golang.org/pkg/os/exec/

[39] "Library logrus documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://github.com/sirupsen/logrus

[40] "Library validator documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: github.com/go-playground/validator/v10

[41] "Library dynamock documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: github.com/gusaul/go-dynamock

[42] "Library assert documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://pkg.go.dev/github.com/stretchr/testify/assert

[43] "Library ishell documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://github.com/abiosoft/ishell

[44] "Library terratest documentation," consultado el 23 de febrero de 2021. [En línea]. Disponible en: https://github.com/gruntwork-io/terratest